
DOCUMENTO DI NOTE

In questo documento vengono riportati i
concetti affrontati durante lo stage presso
Sync Lab s.r.L.

Author

Marco Brugin
Sync Lab s.r.L.
August 7, 2023

Contents

1	Streaming ad eventi	7
1.1	Utilizzi	7
2	Apache Kafka	7
2.1	Utilizzo	7
2.2	Funzionamento	7
2.2.1	Server	7
2.2.2	Client	8
2.3	Garanzie di funzionamento	8
2.4	Gestione degli eventi	8
2.5	Interfacce presenti	8
2.5.1	Kafka Producer	8
2.5.2	Kafka Consumer	8
2.5.3	Kafka Connect	8
2.6	Replicas	9
2.7	Retention	9
2.7.1	Retention basata sul tempo	9
2.7.2	Retention basata sulle dimensione	9
2.8	Zookeeper	10
2.9	Kraft	10
2.10	Casi d'uso	11
2.10.1	Notifica dell'evento	11
2.10.2	Trasferimento di stato portato da eventi	11
2.10.3	Approvvigionamento di eventi	11
2.11	Applicazioni	11
2.11.1	Messaggistica	11
2.11.2	Monitoraggio di siti web	11
2.11.3	Metrica	11
2.11.4	Aggregazione di registri	11
2.11.5	Elaborazione del flusso	11
2.11.6	Approvvigionamento di eventi	12
2.11.7	Registro commit	12
2.12	Apache Kafka VS EDA	12
3	Even Driven Architecture	12
3.1	Casi d'uso	12
3.2	Integrazioni esterne	12
3.3	Flussi di lavoro	13
3.4	Trasferimento di Stato	13
3.5	Accoppiamento temporale	13
4	Middleware	13

5	Panoramica design pattern architetturali	13
5.1	Layered-Architecture	13
5.2	Client-server	13
5.3	Pipe-filter	14
5.4	Broker	14
6	Publisher-Subscriber	14
6.1	Architettura	14
6.2	Vantaggi	14
7	Streaming Data pipelines	15
7.1	Architettura	15
7.2	Obiettivi	15
7.3	Alta affidabilità	16
8	Architetture distribuite	16
8.1	Confronto tra architetture centralizzate e distribuite	16
9	Apache Druid	16
9.1	Caratteristiche principali	17
9.2	Funzionamento	17
9.2.1	DistribuzioniSu singolo nodo	17
9.2.2	Distribuzione in cluster	18
9.3	L'architettura	18
9.4	Servizi offerti	18
9.5	Server principale	18
9.5.1	Servizio di coordinamento	18
9.5.2	Servizio di Overlord	19
9.6	Server di interrogazione	19
9.6.1	Servizio Broker	19
9.6.2	Servizio di router (opzionale)	19
9.7	Server dati	19
9.7.1	Servizio storico	19
9.7.2	Servizio MiddleManager	19
9.7.3	Servizio Peone	19
9.7.4	Servizio di indicizzazione	20
9.8	Deep dive	20
9.8.1	Deep storage	20
9.8.2	Metadata storage	20
9.8.3	Zookeeper	21
9.9	Archiviazione dei dati	21
9.10	I segmenti	22
9.10.1	Struttura del file di segmento	22
9.10.2	L'identificazione del segmento	22
9.10.3	Indicizzazione e consegna del segmento	22
9.10.4	Indentificatore di segmento	23

9.10.5	Versione del segmento	23
9.10.6	Ciclo di vita del segmento	23
9.10.7	Injection dei dati	24
9.11	Casi d'uso	24
9.11.1	Potenza di analisi dati in real-time e applicazioni dati	24
9.11.2	Attività e comportamento dell'utente	24
9.11.3	Flussi di rete	24
9.11.4	Marketing digitale	25
9.11.5	Gestione delle prestazioni dell'applicazione	25
9.11.6	Metriche IoT e dispositivi	25
9.11.7	Operazioni OLAP	25
9.12	Modello dei dati	25
9.12.1	Metrica	26
9.12.2	Dimension	26
9.13	Schema design tips	26
9.13.1	Modello relazionale	26
9.13.2	Modello su serie storiche	26
9.13.3	Utilizzo degli Sketches	27
9.13.4	Utilizzo del timestamp	27
9.13.5	Autorillevamento del tipo	27
9.14	Data rollup	27
9.14.1	Casi d'uso	27
9.14.2	Tipi di rollup	28
9.15	Le lookups	28
9.15.1	Sintassi SQL di query con l'utilizzo delle tabelle di lookup	28
9.15.2	Tipi di lookup	29
9.15.3	Casi d'uso	29
9.16	Confronto Apache Druid e PostgreSQL	29
9.16.1	Query	29

10 Kubernetes 30

10.1	Panoramica e caratteristiche principali	30
10.2	Casi d'uso	30
10.3	Funzionalità	30
10.4	Architettura	31
10.4.1	I componenti del control pane	31
10.4.2	I componenti del nodo	32
10.5	Il funzionamento	32
10.5.1	Unicità del nodo	33
10.5.2	Lo stato di un nodo	33
10.6	Heartbeats	33
10.7	Il controllore del nodo	34
10.8	L'elezione del capo	34
10.9	Garbage Collection	34
10.10	Foreground cascading deletion	34

10.11	I service	35
10.12	Il ConfigMap	35
10.13	Configurazione con minikube	35
10.13.1	Svolgimento	35
10.14	Configurazione attraverso file kubeconfig	38
10.14.1	Composizione	38
10.15	Esempio di deployment di minikube	39
10.15.1	Configurazione l'accesso a più cluster	39
10.16	Esempio di deployments con Apache Druid	42
11	Implementazione	46
11.1	Apache Kafka	46
11.2	Creazione di un generatore e consumatore di eventi in python con Apache Kafka	48
11.2.1	Svolgimento	48
11.3	Creazione di una data pipeline con Apache Kafka, Apache Druid e Docker Compose	51
11.3.1	Svolgimento	51
11.3.2	Confronto tra Apache Druid e Postgres	60
11.3.3	Confronto tra Apache Druid con e senza rollup	64
11.3.4	Utilizzo delle tabelle di lookup in Apache Druid	67
12	Test	69
12.1	High Availability	69
12.1.1	Svolgimento	69
12.2	Verifica di connessione al container da un altro host	70
12.2.1	Svolgimento	70

List of Figures

1	Confronto tra architetture centralizzate e distribuite	16
2	Architettura Druid	18
3	get deployment	36
4	Output PODS	36
5	Get events	37
6	Config view	37
7	Cluster Kafka	50
8	Iniezione dei dati	51
9	Load data - Streaming-Apache kafka	59
10	Attivazione della funzionalità di rollup	66
11	Creazione della tabella di lookup	68
12	Risultato della query utilizzando la tabella di lookup	69
13	Creazione del cluster	69
14	Creazione del topic	70
15	Creazione del produttore	70
16	Creazione del consumatore	70
17	Test di alta affidabilità	70
18	Creazione del cluster	71
19	Creazione dei topic	71
20	Connessione da altro host	71

1 Streaming ad eventi

È una pratica di acquisizione dei dati in tempo reale da fonti di eventi come database, flussi di eventi; memorizzando tutto ciò per un recupero futuro di tali informazioni, reagendo a flussi di eventi in tempo reale. Inoltre garantisce un flusso continuo di info corrette nel posto giusto e al momento giusto.

1.1 Utilizzi

- per transazioni
- per servizi IOT di vario genere
- monitoraggio sanitario
- ovunque ci sia la necessità di trattare grandi moli di dati efficientemente

2 Apache Kafka

Kafka è una piattaforma open source che combina 3 funzionalità in modo da poter soddisfare i casi d'uso sopra citati:

- pubblica e sottoscrive flussi di eventi, importandoli ed esportandoli da altri sistemi;
- archivia tali flussi in modo affidabile e duraturo;
- elabora flussi di eventi in real time o in modo retrospettivo.

2.1 Utilizzo

Kafka opera su una architettura distribuita. Può essere distribuito e utilizzato in vari modi tra cui virtual machine e container, on-promise, o servizi cloud.

2.2 Funzionamento

Kafka nasce come sistema distribuito che opera su nodi che comunicano tramite protocollo **TCP** ad alte prestazioni. Data la sua natura distribuita implementa capacità di fault tolerance con rimpiazzo dei nodi guasti. **Kafka** è costituito da due componenti essenziali: server e client.

2.2.1 Server

Kafka viene eseguito come un cluster di uno o più server. Alcuni fanno da **broker**: livello di archiviazione. Altri assolvono il compito di **Kafka Connect**: importano e esportano i dati sotto forma di flussi di eventi che permette di interagire con altri sistemi esistenti.

2.2.2 Client

Consentono di scrivere applicazioni distribuite e microservizi che leggono, scrivono ed elaborano flussi di eventi in parallelo, su larga scala e con fault tolerance anche in caso di problemi di rete o guasti della macchina. Esistono molti client per diversi linguaggi di programmazione.

2.3 Garanzie di funzionamento

In **Kafka** esistono produttori e consumatori che producono e sottoscrivono eventi. Gli uni sono indipendenti l'uno dall'altro, ciò permette di raggiungere alcune delle seguenti garanzie:

- **Al massimo una volta:** i messaggi possono andare persi ma mai riconsegnati. Infatti si potrebbe verificare un guasto ad un singolo broker o un errore della rete nel momento dell'invio di un messaggio. Il messaggio stesso verrà recapitato una e una sola volta.
- **Almeno una volta:** i messaggi non vengono mai persi ma possono essere riconsegnati;
- **Una sola volta:** i messaggi non vanno persi e sono consegnati una sola volta; è la garanzia maggiormente desiderabile.

2.4 Gestione degli eventi

Gli eventi sono organizzati in topic o argomenti. Gli eventi possono essere letti da più consumatori. Un evento anche se consumato non viene eliminato, può essere impostato un timeout di mantenimento dell'evento. I topic sono partizionati su più nodi. Tale partizionamento consente ai client di leggere e scrivere dati da/a molti broker. Quando un nuovo evento viene emesso questo si aggiunge alla rispettiva partizione. Sarà **Kafka** a garantire che gli eventi vengano letti nell'ordine in cui sono stati scritti.

2.5 Interfacce presenti

2.5.1 Kafka Producer

L'interfaccia Producer permette alle applicazioni di inviare i flussi di dati ai broker di un cluster di Apache per categorizzarli e salvarli (nei topic già citati).

2.5.2 Kafka Consumer

L'interfaccia Consumer consente ai consumatori di Apache **Kafka** di ricevere l'accesso ai dati, salvati nei topic di un cluster.

2.5.3 Kafka Connect

L'interfaccia Connect consente di impostare produttori e consumatori riutilizzabili che colleghino i topic di **Kafka** con le applicazioni o le banche dati esistenti.

2.6 Replicas

Con il termine di **Kafka Replication** significa disporre di più copie dei dati, distribuite su più server/broker. Ciò aiuta a mantenere alti livelli di disponibilità in caso di guasto.

Le repliche sono permesse a livello di partizione. kafka ne designa una chiamata leader mentre le altre sono partizioni follower o **in-sync**. Il numero totale di repliche incluso il leader costituisce il fattore di replicazione. Il leader è responsabile della ricezione e dell'invio dei dati, per quella partizione. Per mantenere questi cluster e gli argomenti/partizioni all'interno, Kafka ha un servizio centralizzato chiamato **Zookeeper**.

2.7 Retention

Con **Retention** in **Kafka** si intende la possibilità di controllare la dimensione dei registri degli argomenti ed evitare di superare le dimensioni del disco esistente.

La conservazione può essere configurata o controllata in base alla dimensione dei log o in base alla durata configurata. Tale configurazione può essere impostata a grana fine o a grana grossa per ogni argomento o per tutti gli argomenti.

2.7.1 Retention basata sul tempo

Una volta raggiunto il tempo di conservazione configurato per il segmento, questo viene contrassegnato per l'eliminazione o la compattazione in base al criterio di pulizia configurato. Il periodo di conservazione predefinito per i segmenti è di 7 giorni. In ordine di importanza i parametri configurabili per la **Retention** basata sul tempo sono in ordine di importanza e valutazione:

1. log.retention.ms;
2. log.retention.minutes;
3. log.retention.hours.

Nel momento in cui un parametro di livello di priorità superiore non è impostato si segue ciò che indica quello di livello appena inferiore.

2.7.2 Retention basata sulle dimensione

In questo caso si va configurare la dimensione massima di una struttura di dati di registro per una partizione di argomento. Una volta che la dimensione del registro raggiunge questa dimensione, inizia a rimuovere i segmenti dalla sua fine.

In ordine di importanza i parametri configurabili per la **Retention** basata dimensione sono in ordine di importanza e valutazione:

1. log.segment.bytes: la dimensione massima di un singolo file di registro;
2. log.retention.check.interval.ms: la frequenza in millisecondi con cui la pulizia del registro verifica se un registro è idoneo per l'eliminazione;
3. log.segment.delete.delay.ms: la quantità di tempo da attendere prima di eliminare un file dal file system.

2.8 Zookeeper

Zookeeper è un prodotto open-source che si occupa della sincronizzazione tra i cluster distribuiti e ne gestisce le configurazioni, il controllo e la denominazione.

Il protocollo **Zookeeper Atomic Broadcast (ZAB)** è il cervello dell'intero sistema.

Ogni replica o nodo invia a intervalli regolari un messaggio *Keep-Alive* a Zookeeper, informando così **Zookeeper** che è vivo e funzionante.

Se entro un tempo prestabilito il messaggio non viene ricevuto si presume che il nodo sia morto e se era un leader se ne elegge un altro.

Zookeeper permette di definire dei parametri che consentono di capire quando un nodo è guasto e quanto i nodi follower sono in ritardo rispetto al leader.

Il parametro `zookeeper.session.timeout.ms` millisecondi è impostato su 6000 per impostazione predefinita: indica che, se il leader non riceve l'evento *Keep-Alive* entro quel periodo temporale, detiene che quel nodo sia morto.

Il parametro `replica.lag.max.messages`, decide la differenza consentita tra **Replica's Offset** e **Leader's Offset**. Se questa differenza è maggiore di `replica.lag.max.messages-1`, il nodo viene considerato in ritardo e viene rimosso dall'elenco dei nodi in sincronizzazione dal leader.

Tutti i nodi che sono attivi e sincronizzati formano l' **In-Sync Replica Set (ISR)**.

Ora, se tutti i nodi in sincronizzazione hanno applicato un messaggio ai rispettivi log, questo messaggio viene considerato confermato e quindi inviato ai consumatori. In questo modo, **Kafka** garantisce che un messaggio di cui è stato eseguito il commit non andrà perso, purché sia presente almeno una replica attiva e sincronizzata, in ogni momento.

Un nodo non sincronizzato può ricongiungersi all'**ISR** se può risincronizzarsi completamente di nuovo, anche se ha perso alcuni dati a causa del suo arresto anomalo.

2.9 KRaft

Apache Kafka Raft (KRaft) è il protocollo di consenso introdotto per rimuovere la dipendenza di **Apache Kafka** da **ZooKeeper** per la gestione dei metadati. Ciò semplifica enormemente l'architettura di **Kafka** consolidando la responsabilità dei metadati all'interno di **Kafka** stesso, anziché suddividerla tra due diversi sistemi: **ZooKeeper** e **Kafka**.

KRaft utilizza un modello di archiviazione di origine evento che garantisce che le macchine a stati interni possano sempre essere ricreate accuratamente. Il registro eventi utilizzato per archiviare questo stato viene periodicamente abbreviato da istantanee per garantire che il registro non possa crescere all'infinito.

Gli altri nodi all'interno del quorum seguono il leader rispondendo agli eventi che crea e memorizza nel suo registro. Pertanto, se un nodo si ferma a causa di un evento di partizionamento, ad esempio, può recuperare rapidamente eventuali eventi persi accedendo al registro quando si ricongiunge.

La natura basata sugli eventi del protocollo KRaft significa che, a differenza del leader basato su ZooKeeper, il leader del quorum non ha bisogno di caricare lo stato da ZooKeeper prima che diventi attivo. Quando la leadership cambia, il nuovo leader attivo ha già tutti i record di metadati impegnati in memoria. Inoltre, lo stesso meccanismo guidato dagli eventi utilizzato nel protocollo KRaft viene utilizzato per tenere traccia dei metadati nel cluster.

2.10 Casi d'uso

2.10.1 Notifica dell'evento

Kafka permette di trasmettere semplicemente eventi per avvisare altri sistemi di un cambiamento nel suo dominio.

2.10.2 Trasferimento di stato portato da eventi

In questo utilizzo il destinatario dell'evento ottiene anche i dati di cui ha bisogno per eseguire azioni aggiuntive sui dati richiesti.

2.10.3 Approvvigionamento di eventi

In questo caso il sistema consente di descrivere ogni cambiamento di stato in un sistema come un evento, con ogni evento registrato in sequenza cronologica. Di conseguenza, il flusso di eventi stesso diventa la principale fonte di verità del sistema.

2.11 Applicazioni

2.11.1 Messaggistica

I broker di messaggi vengono utilizzati per disaccoppiare l'elaborazione del messaggio dal produttore dello stesso. **Kafka** rispetto ai sistemi di messaggistica tradizionale ha una migliore velocità, partizionamento integrato e tolleranza agli errori. L'utilizzo è basso, ma in ambiti in cui si richiede una bassa latenza, le garanzie fornite da **Kafka** sono alla pari dei tradizionali sistemi di messaggistica, che lo rendono una buona soluzione per applicazioni di elaborazione di messaggi su larga scala.

2.11.2 Monitoraggio di siti web

È il caso d'uso d'origine di **Kafka**, ha la caratteristica di generare un elevato volume di messaggi. Lo scopo era quello di ricostruire le attività degli utenti come insieme di eventi generati dalle azioni dell'utente.

2.11.3 Metrica

Indica l'aggregazione di dati di monitoraggio provenienti da varie fonti per eseguire statistiche.

2.11.4 Aggregazione di registri

Kafka ha anche la capacità di estrarre i dati dai file di log fisici e fornisce una astrazione di tali sotto forma di flusso di messaggi. Ciò permette di garantire una più bassa latenza di elaborazione e supporto per più origini dati.

2.11.5 Elaborazione del flusso

È possibile andare a creare una data-pipeline in cui i dati grezzi vengono consumati dagli argomenti di **Kafka**, aggregati, trasformati fino ad ottenere un dato elaborato.

2.11.6 Approvvigionamento di eventi

È possibile utilizzare di **Kafka** in applicazioni cui cambiamenti di stato vengono registrati come sequenze di record in ordine temporale. **Kafka** permette il supporto per dati di registro memorizzati molto grandi tanto che lo rende un ottimo back-end di tali applicazioni.

2.11.7 Registro commit

Kafka può fungere da sorta di log di commit esterno per un sistema distribuito. Il registro aiuta a replicare i dati tra i nodi e funge da meccanismo di risincronizzazione per consentire ai nodi non allineati di ripristinare i propri dati.

2.12 Apache Kafka VS EDA

L'emissione di un evento indica che qualcosa è accaduto e può essere visto come un agglomerato di dati atomico in grado di soddisfare l'evento stesso. **Kafka** è un sistema di streaming di eventi che gestisce un flusso continuo di eventi. Inoltre Kafka memorizza tali in modo duraturo per il successivo recupero, analisi o elaborazione in tempo reale e li inoltra a varie destinazioni secondo necessità.

D'altra parte in una **EDA** (Event Driven Architecture) viene generati degli eventi che un agente acquisisce e risponde a tale evento. per utilizzare **Kafka** in un sistema **EDA** la chiave è andare a sfruttare il disaccoppiamento: invece di effettuare un polling continuo di verifica della presenza di nuovi dati, basterà ascoltare il verificarsi di un evento per agire. Inoltre grazie all'approccio sviluppato da **Kafka** un evento una volta soddisfatto non viene eliminato, ma conservato per un periodo di tempo predeterminato, pertanto un evento potrà essere letto da più consumatori e potrà essere utilizzato per soddisfare una varietà di richieste.

3 Even Driven Architecture

L'**Even Driven Architecture** è un pattern architetturale basato su eventi: degli agenti, che sono in grado di ricevere tali eventi, agiranno solo nel momento in cui questi ultimi si verificheranno. Una architettura basata su eventi fornisce una serie di altri vantaggi basati sul disaccoppiamento tra il produttore e il consumatore dell'evento, i quali, nel momento dell'emissione di quest'ultimo non è necessario siano sincroni ma possono andare a sfruttare una comunicazione di tipo asincrona.

3.1 Casi d'uso

3.2 Integrazioni esterne

In molti sistemi quando si verifica un evento è necessario interagire con un servizio esterno. Grazie ad una architettura ad eventi, ogni servizio sarà indipendente l'uno dall'altro, così che se un servizio riscontra problemi nella sua esecuzione gli altri non ne risentiranno, data la loro indipendenza. Inoltre anche l'aggiunta di nuovi servizi ha nessun impatto.

3.3 Flussi di lavoro

L'architettura **EDA** può essere anche impiegata nell'orchestrazione di un flusso di lavoro; nel quale ogni servizio è responsabile di un part del flusso indipendentemente dagli altri. Il fattore chiave è andare ad eseguire il rispettivo servizio all'emissione di un determinato evento.

3.4 Trasferimento di Stato

La modifica di un dato è un altro caso d'uso in cui l'architettura **EDA** può essere utilizzato per segnalare in tempo reale il mutamento di un dato in una determinata struttura dati.

3.5 Accoppiamento temporale

Inoltre grazie ad un' architettura **EDA** si va a rimuovere l'accoppiamento temporale tra chi effettua una chiamata e chi va a risponde a quest'ultima.

4 Middleware

Con il termine di **Middleware** si va ad intendere un insieme di pacchetti software che molte applicazioni utilizzano per comunicare tra di loro. Il **Middleware** funge da ponte tra tecnologie differenti in modo da poterle totalmente integrare.

L'architettura sottostante ad un **Middleware** non è altro che una **data-pipeline** nella quale i dati passano da una applicazione in connessione all'altra.

La chiave di utilizzo di un **Middleware** è che fornisce disaccoppiamento tra i vari stadi di una **data-pipeline**.

5 Panoramica design pattern architetturali

5.1 Layered-Architecture

È un modello architetturale basato su più livelli orizzontali, in cui ciascuno svolge un compito specifico e ha determinata responsabilità. Sebbene il pattern in sé non indichi il numero di livelli da utilizzare, è usuale creare architetture con al più quattro livelli: presentazione, business, persistenza e database. In tale architettura ogni livello è del tutto separato dagli altri: le modifiche apportate ad un livello non generano cambiamenti agli altri, per questo si parla di livelli isolati. La comunicazione tra i livelli avviene in dall'alto verso il basso e in generale attraverso livelli chiusi: il passaggio delle informazione tra due o più livelli deve attraversare tutti i livelli intermedi.

5.2 Client-server

Nel pattern architetturale client-server ci sono due componenti essenziali: il client, che richiede un servizio o una prestazione, e il server che la fornisce. Il client espone porte di richiesta, mentre il server espone porte di servizio. In tale architettura vige un tipo di collegamento Request-Response. Uno dei principali vantaggi di tale pattern architetturale il calcolo centralizzato dei dati.

5.3 Pipe-filter

Il modello architetturale **Pipe-Filter** è caratterizzato da trasformazioni successive del flusso di dati.

I dati dall'origine vengono inviati dall'origine verso le porte di input del primo filtro, dove viene eseguita la prima elaborazione, quindi il prodotto dell'elaborazione viene fornito dall'output all'input del filtro successivo e così via.

Il fattore importante in tale pattern architetturale è l'ordine con cui vengono eseguiti tali filtri.

5.4 Broker

Il **Broker** pattern architetturale che può essere utilizzato per sistemi software disaccoppiati e distribuiti che interagiscono tramite chiamate remote. Il pattern è formato da tre agenti:

- il client: che accede a funzionalità del server inviando le richieste al broker;
- il server che mette a disposizione del client i propri servizi attraverso il broker;
- il broker: riceve le richieste dai client, le archivia e una volta trovato il server appropriato inoltra la richiesta e trasmette il risultato al client.

6 Publisher-Subscriber

Il pattern architetturale **Publisher-Subscriber** è un modello di progettazione software, utilizzato nei sistemi distribuiti, che impiegano una comunicazione asincrona tra i vari componenti.

Sebbene vada ad utilizzare tecniche già preesistenti come la sottoscrizione e l'accodamento di messaggi, la chiave di successo di tale pattern è il totale disaccoppiamento delle componenti: i componenti non sono a conoscenza dell'identità e della presenza degli altri.

Il modello Pub-SUB è nato dalla necessità del rendere i sistemi ridimensionabili in modo dinamico.

6.1 Architettura

Il Pub-Sub fornisce un sistema di scambio di messaggi tra editori e sottoscrittori. Lo scambio tra questi ultimi non avviene in modo diretto tra gli agenti appena illustrati, ma viene utilizzato un intermediario che raggruppa i messaggi per argomento e fornisce disaccoppiamento tra le componenti.

6.2 Vantaggi

- **Debole accoppiamento tra le componenti:** che rende il sistema più flessibile e modificabile dinamicamente;
- **Elevata scalabilità:** non esiste limite al numero di publisher e subscriber che possano comunicare;

- **Utilizzo della comunicazione asincrona ad eventi:** non necessità della sincrona degli attori coinvolti nella comunicazione;
- **Indipendente dal protocollo di comunicazione:** è integrabile con qualsiasi protocollo di comunicazione e stack tecnologico.

7 Streaming Data pipelines

Con il termine di **data-pipelines** si intende un software che consente il fluire automatico di dati da un punto ad un altro del sistema. Di solito le origini sono molteplici e generano dati ad altissima velocità.

L'architettura che sta dietro a una **data-pipeline** consente di consumare, elaborare, archiviare dati in tempo reale, man mano che vengono generati. Tutto ciò consente analisi e reazioni più veloci ad esigenze sorte.

In generale nelle pipeline dati tradizionali estraggono, trasformano e caricano i dati prima che possano essere utilizzati. Ma data la enorme mole di dati che le **Streaming Data pipelines** sono sottoposte, tutto ciò non è possibile.

7.1 Architettura

Infatti per elaborare dati in streaming provenienti da sistemi come **Kafka** è necessario creare due livelli per l'elaborazione dati:

- **Archiviazione:** questo livello consente la memorizzazione dei dati, permettendo letture e scritture a basso costo in termini computazionali mantenendo l'ordine di arrivo dei dati;
- **Lavorazione:** elabora e consuma i dati del livello di archiviazione andando a segnalare a quest'ultimo i dati non più necessari.

7.2 Obiettivi

- **Scalabilità:** il volume dei dati può crescere notevolmente nel tempo quindi è necessario archiviare i dati in un **data warehouse**;
- **Durata e consistenza:** è importante considerare che i dati letti potrebbero essere già modificati o obsoleti, quindi è fondamentale disporre di strumenti di monitoraggio del flusso dati nel tempo;
- **Ordinare:** l'architettura deve essere in grado di individuare la sequenza dei dati nel flusso;
- **Tolleranza ai guasti:** lo streaming dei dati non smettono mai di fluire da varie sorgenti e il sistema deve prevenire le interruzioni di alcuni sorgenti;
- **Latenza:** data l'enorme mole di dati prodotti, se questi ultimi non vengono gestiti in tempo reale perdono di rilevanza e diventano inutilizzabili.

7.3 Alta affidabilità

Il concetto di alta affidabilità è essenzialmente alla capacità di un sistema di funzionare continuamente senza guasti in un determinato periodo di tempo.

Per garantire tali prestazioni in generale è necessario seguire i seguenti principi:

- **Crossover affidabile:** creazione di ridondanza eseguendo l'atto di passare dal componente X al componente Y senza perdere dati o influire sulle prestazioni;
- **Bilancio del carico:** tecnica attraverso la quale si va a ridistribuire il carico di elaborazione da su tutte le risorse del sistema in modo tale che nessuna sia sovraccaricata;
- **Rilevazione e gestione automatica dei guasti:** i guasti devono essere visibili e i sistemi devono disporre di un sistema deve essere provvisto di un sistema automatico in grado di gestirli autonomamente.

8 Architetture distribuite

Con il termine **distribuito** si va ad intendere un'architettura nella quale i componenti non si trovano su un'unica macchina ma possono cooperare attraverso una rete di comunicazione per raggiungere un determinato obiettivo.

Le principali caratteristiche che si riscontrano su tale architettura sono:

- l'elaborazione non è limitata a una sola macchina;
- Il middleware è un'infrastruttura che supporta adeguatamente lo sviluppo e l'esecuzione di applicazioni distribuite, fornendo un buffer tra le applicazioni e la rete. Svolge il ruolo di intermediario del sistema e gestisce o supporta i diversi componenti di un sistema distribuito;
- le basi di un'architettura distribuita sono la trasparenza, affidabilità e disponibilità.

8.1 Confronto tra architetture centralizzate e distribuite

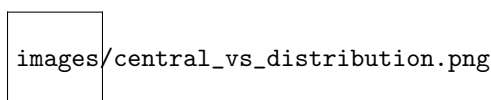


Figure 1: Confronto tra architetture centralizzate e distribuite

9 Apache Druid

Apache Druid è un database di analisi in tempo reale progettato per analisi rapide, i suoi punti di forza sono le alte prestazioni in esecuzione di query su set di dati di grandi dimensioni. **Druid** è comunemente utilizzato come back-end del database per le GUI di applicazioni analitiche o per API altamente simultanee che richiedono aggregazioni veloci. Mostra le sue migliori prestazioni con dati orientati agli eventi.

9.1 Caratteristiche principali

Druid combina le idee provenienti da serie temporali e sistema di ricerca, *data-warehouse*. Le caratteristiche principali sono:

- formato di archiviazione a colonne: utilizza una archiviazione orientata alle colonne e carica soltanto le colonne necessarie per una determinata query. Inoltre ottimizza l'archiviazione delle colonne in base al tipo di dato;
- sistema distribuito scalabile: le distribuzioni tipiche di Druid si estendono su cluster che vanno da decine a centinaia di server ciò permette di mantenere un tempo di latenza molto basso, al di sotto di pochi secondi;
- elaborazione massicciamente parallela: le query vengono elaborate in parallelo sull'intero cluster;
- injection in tempo reale o batch: **Druid** è in grado di elaborare sia dati in tempo reale che in batch, i dati acquisiti sono immediatamente disponibili per le successive interrogazioni;
- è dotato di un sistema di autobilanciamento e auto individuazione dei guasti: il cluster Druid si riequilibra automaticamente in background senza tempi di inattività, inoltre se un server **Druid** si guasta, il sistema instrada automaticamente i dati intorno al danno fino alla risoluzione, infatti **Druid** è progettato per funzionare senza tempi di inattività;
- l'architettura nativa del cloud è tollerante ai guasti: **Druid** archivia in modo sicuro una copia dei tuoi dati in un archivio profondo (in genere su file system o sul cloud), tali dati saranno accessibili sempre anche nel momento in cui i server falliscano; ciò permette l'esecuzione di query anche nei momenti di ripristino.
- Indici per filtraggio rapido: **Druid** utilizza indici bitmap compressi per creare indici che consentano il filtraggio rapido e la ricerca su più colonne;
- Partizionamento basato sul tempo: **Druid** prima suddivide i dati in base al tempo, è possibile in ogni caso aggiungere altri tipi di filtri;
- Algoritmi approssimati: **Druid** permette l'utilizzo di algoritmi offrono un utilizzo limitato della memoria e sono spesso sostanzialmente più veloci dei calcoli esatti, dove sono richiesti conteggi più esatti **Druid** offre pure questa funzionalità.
- Riepilogo automatico al momento dell'acquisizione: **Druid** supporta facoltativamente il riepilogo dei dati al momento dell'acquisizione.

9.2 Funzionamento

9.2.1 DistribuzioniSu singolo nodo

Apache Druid include molteplici configurazioni su singola macchina si trattano di distribuzioni oramai poco utilizzate.

Esistono configurazioni molto piccole pensate per macchine con poca **CPU** e **memoria**, pensate per ambienti con risorse limitate, come i piccoli contenitori *Docker*.

9.2.2 Distribuzione in cluster

Apache Druid è progettato per essere distribuito come cluster scalabile e con tolleranza ai guasti.

In generale è consigliato andare a creare un cluster che ospita:

- i server principali (Coordinator e Overlord): sono responsabili della gestione dei meta-dati e delle esigenze di coordinamento del cluster, possono essere collocati insieme sullo stesso server;
- i server dati (Historicals e MiddleManager): per gestire i dati effettivi nel tuo cluster, traggono grandi vantaggi da CPU, RAM e SSD;
- i server di interrogazione: i **Druid Broker** accettano le richieste e le distribuiscono al resto del cluster, è possibile anche mantenere una cache delle query in memoria.

9.3 L'architettura

Druid ha un'architettura distribuita progettata per essere compatibile con il cloud. È possibile andare ridimensionare i servizi per avere la massima flessibilità. Inoltre una architettura di questo tipo è maggiormente tollerante ai guasti, un malfunzionamento di una componente non influisce immediatamente sulle altre.

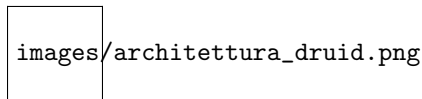


Figure 2: Architettura Druid

9.4 Servizi offerti

9.5 Server principale

Un server Master gestisce l'ingestion e la disponibilità dei dati: è responsabile dell'avvio di nuovi job di ingestion e del coordinamento della disponibilità dei dati sui "Data server" descritti di seguito.

9.5.1 Servizio di coordinamento

Ha il compito di gestire la disponibilità dei dati del cluster. **Druid Coordinator** comunica ai processi storici di caricare o rilasciare segmenti in base alle configurazioni. Inoltre è responsabile del caricamento di nuovi segmenti, dell'eliminazione di segmenti obsoleti, della garanzia che i segmenti siano "replicati" (ovvero, caricati su più nodi storici diversi) un numero corretto (configurato) di volte e dello spostamento ("bilanciamento") dei segmenti tra Historical nodi per mantenere quest'ultimo caricato uniformemente. **Druid Coordinator** prima di eseguire le varie operazioni valuta lo stato attuale del cluster. Inoltre come gli altri servizi mantiene una connessione a un cluster **Zookeeper** per le informazioni sul cluster corrente. Infine il servizio di coordinamento ha anche una connessione ad un database contenente l'elenco dei segmenti utilizzati.

9.5.2 Servizio di Overlord

È responsabile dell'accettazione delle attività, del coordinamento della distribuzione delle attività, della creazione di blocchi attorno alle attività e della restituzione degli stati ai chiamanti. È possibile eseguirlo sia in modalità locale, per flussi di lavoro semplici, che remota nella quale occupa un server dedicato.

9.6 Server di interrogazione

Un server di query fornisce gli endpoint con cui interagiscono gli utenti e le applicazioni client, instradando le query ai server di dati o ad altri server di query.

9.6.1 Servizio Broker

In un cluster distribuito è il servizio a cui instradare le query. Comprende i metadati forniti da **Zookeeper**, dove ricavare i dati necessari ad elaborare i risultati delle query. Infine unisce i risultati di tutti i risultati elaborati per fornire un'unica risposta alla query richiesta.

9.6.2 Servizio di router (opzionale)

È il servizio che ha il compito di instradare le query a diversi processi **Broker**. Le query vengono instradate in base alle regole di caricamento dei segmenti che vengono applicate. Questa configurazione fornisce l'isolamento delle query in modo tale che le query per i dati più importanti non siano influenzate dalle query per i dati meno importanti.

9.7 Server dati

Un server di dati esegue processi di acquisizione e archivia dati interrogabili.

9.7.1 Servizio storico

Ogni servizio copia o estrae i file di segmento da Deep Storage al disco locale in un'area chiamata *segment cache* e rispondono alle domande su tali segmenti.

Il coordinatore controlla l'assegnazione dei segmenti agli storici e l'equilibrio dei segmenti tra gli storici. I servizi di storico non comunicano direttamente tra loro, né comunicano direttamente con il Coordinatore. Invece, il coordinatore crea voci temporanee in **Zookeeper** in un percorso della coda di caricamento. Ogni processo storico mantiene una connessione con **Zookeeper**, osservando quei percorsi per le informazioni sui segmenti.

9.7.2 Servizio MiddleManager

I processi MiddleManager gestiscono l'inserimento di nuovi dati nel cluster. Sono responsabili della lettura da fonti di dati esterne e della pubblicazione di nuovi segmenti Druid.

9.7.3 Servizio Peone

I servizi Peon sono motori di esecuzione delle attività generati dai **MiddleManager**. Ogni **Peon** esegue una JVM separata ed è responsabile dell'esecuzione di una singola attività. I Peon girano sempre sullo stesso host del **MiddleManager** che li ha generati.

9.7.4 Servizio di indicizzazione

In alternativa ai **MiddleManger** ed ai **Peon**.

Invece di eseguire il fork di processi JVM separati per attività, l'indicizzatore esegue le attività come singoli thread all'interno di un singolo processo JVM.

L'indicizzatore è progettato per essere più facile da configurare e distribuire rispetto al sistema MiddleManager + Peon e per consentire una migliore condivisione delle risorse tra le attività.

9.8 Deep dive

9.8.1 Deep storage

Il **deep storage** è dove vengono archiviati i segmenti. È un meccanismo non fornito da **Druid**.

Fintanto i servizi di **Druid** possono vedere questa infrastruttura di archiviazione e accedere ai segmenti archiviati su di essa, non si perdereanno i dati indipendentemente dal numero di nodi Druid che persi.

Il **deep storage** può essere di due tipi:

- locale: destinato ad un solo server o a più server che hanno accesso ad un filesystem condiviso;
- su cloud: è più conveniente, più scalabile e più robuste rispetto all'impostazione di un filesystem condiviso. **Druid** è compatibile con diversi sistemi cloud come: Azure, Amazon S3, Google.

9.8.2 Metadata storage

È l'archivio, basato su dipendenza esterna, dove vengono conservati tutti i metadati essenziali per il funzionamento di un cluster Druid. Druid utilizza l'archivio dei metadati per ospitare vari metadati sul sistema, ma non per archiviare i dati effettivi.

Contiene:

- record di segmenti;
- record di regole;
- record di configurazione;
- registri di controllo.

Come servizio predefinito **Druid** utilizza **Derby**, ma archivi più adatti alla produzione sono **MySQL** e **PostgreSQL**.

Tabella dei segmenti Questa tabella memorizza i metadati sui segmenti che dovrebbero essere disponibili nel sistema (chiamati anche segmenti usati). La tabella viene interrogata dal coordinatore per determinare l'insieme di segmenti che dovrebbero essere disponibili per l'interrogazione nel sistema. La tabella ha due colonne funzionali principali, le altre colonne sono a scopo di indicizzazione.

Il primo valore **used** indica che il segmento deve essere utilizzato dal cluster, è il mezzo

attraverso cui si caricano e scaricano i segmenti dal cluster senza rimuovere i metadati. La seconda è la **payload** che archivia un BLOB JSON che contiene tutti i metadati per il segmento.

Tabella delle regole È la tabella che memorizza le varie regole su dove allocare e deallocare i segmenti del cluster.

Tabella di configurazione Hanno lo scopo di memorizzare gli oggetti di configurazione di runtime.

In futuro avranno lo scopo di modificare alcuni parametri di configurazione nel cluster in fase di esecuzione.

Tabella delle attività Si riferiscono alle tabelle create e gestite dall'**Overlord** e **MiddleManager**.

Tabella di revisione È la tabella che memorizza le modifiche apportate alla configurazione.

Hanno accesso ai metadati solo: il servizio di indicizzazione, il servizio in tempo reale e il servizio di coordinamento.

9.8.3 Zookeeper

Apache Druid utilizza **Apache ZooKeeper** per la gestione dello stato corrente del cluster. In **Zookeeper** avvengono le seguenti attività:

- Elezione del servizio di coordinamento;
- Protocollo di "pubblicazione" del segmento degli storici;
- Protocollo di caricamento/rilascio del segmento tra Coordinatore e Storico;
- elezione del servizio di **Overlord**;
- Gestione delle attività di **Overlord** e **MiddleManager**;

9.9 Archiviazione dei dati

I dati in **Apache Druid** vengono archiviati in *datasources* tabelle simili a quelle utilizzate dai database relazionali. Ogni tabella viene partizionata in base agli intervalli di tempo. Ogni intervallo di tempo è chiamato blocco. All'interno di un blocco, i dati vengono partizionati in uno o più segmenti. Ogni segmento è un singolo file, che in genere comprende fino a pochi milioni di righe di dati. Si può pensare che i segmenti di dati è come vivano in una sequenza temporale.

Ogni segmento viene creato da un **MiddleManager** come mutable e uncommitted. I dati sono interrogabili non appena vengono aggiunti a un segmento senza commit.

Il processo di creazione del segmento accelera le query successive producendo un file di dati compatto e indicizzato.

Periodicamente i segmenti vengono sottoposti a commit e pubblicati in deep storage diventano immutabili e passano dal **MiddleManager** ai processi storici. Inoltre nell'archivio dei

metadati viene scritta una voce al segmento che lo descrive.
Tali informazioni indicano al coordinatore quali dati sono presenti nel cluster.

9.10 I segmenti

Apache Druid memorizza i suoi dati e indici in file di segmenti partizionati in base al tempo. **Druid** crea un segmento per ogni intervallo di tempo che contiene dati. Affinchè **Druid** funzioni bene con un carico di query elevato, è importante che la dimensione del file del segmento rientri nell'intervallo consigliato di 300-700 MB. Se questo non accade si può valutare la possibilità di modifica della granularità dell'intervallo di tempo del segmento.

9.10.1 Struttura del file di segmento

I file di segmento sono colonnari : i dati per ciascuna colonna sono disposti in strutture di dati separate. Memorizzando ciascuna colonna separatamente, Druid riduce la latenza delle query analizzando solo le colonne effettivamente necessarie per una query.

Una volta che una query identifica le righe da selezionare, le decompone, estrae le righe pertinenti e applica l'operatore di aggregazione desiderato. Se una query non richiede una colonna, Druid ignora i dati di quella colonna. Le colonne di tipo dimension supportano le operazioni di filtro e raggruppamento, quindi richiede l'utilizzo delle seguenti tre strutture dati:

- Dizionario : esegue il mapping dei valori agli ID interi, consentendo una rappresentazione compatta dell'elenco e dei valori bitmap;
- List : i valori della colonna, codificati utilizzando il dizionario. Questi operatori consentono l'esecuzione di query che aggregano esclusivamente metriche basate su filtri senza accedere all'elenco di valori.
- Bitmap : una bitmap per ogni valore distinto nella colonna, per indicare quali righe contengono tale valore.

Per ogni riga nell'elenco dei dati della colonna, esiste solo una singola bitmap con una voce diversa da zero. Ciò significa che le colonne con cardinalità elevata hanno bitmap estremamente rare e quindi altamente comprimibili.

9.10.2 L'identificazione del segmento

Gli identificatori di segmento in genere contengono l'origine dati del segmento, l'ora di inizio dell'intervallo (in formato ISO 8601 yyyy-mm-gg), l'ora di fine dell'intervallo (in formato ISO 8601 yyyy-mm-gg) e le informazioni sulla versione. Se i dati vengono ulteriormente partizionati oltre un intervallo di tempo, l'identificatore di segmento contiene anche un numero di partizione.

9.10.3 Indicizzazione e consegna del segmento

L'**indicizzazione** è il meccanismo mediante il quale vengono creati nuovi segmenti e l'**handoff** è il meccanismo mediante il quale vengono pubblicati e iniziano a essere serviti dai processi storici.

Un'attività di indicizzazione viene avviata e crea un nuovo segmento. Deve determinare l'identificatore del segmento prima di iniziare a costruirlo.

L'attività di aggiunta di un nuovo segmento viene fatta chiamando un'API di "allocazione" su **Overlord** per aggiungere potenzialmente una nuova partizione a un set di segmenti esistente. Per un'attività che sta sovrascrivendo (come un'attività Hadoop o un'attività di indicizzazione non in modalità di aggiunta), questa operazione viene eseguita bloccando un intervallo e creando un nuovo numero di versione e un nuovo set di segmenti.

Da questo momento in poi il segmento è interrogabile e quando l'attività di indicizzazione ha terminato li inserisce nello deep storage e li pubblica scrivendolo nell'archivio dei metadati. Per i processi in real-time, per garantire la disponibilità dei dati attende il caricamento dei dati.

9.10.4 Identificatore di segmento

Tutti i segmenti hanno un identificatore in quattro parti con i seguenti componenti:

- nome dell'origine dati;
- intervallo di tempo;
- numero di versione;
- numero di partizione.

9.10.5 Versione del segmento

Il numero di versione fornisce una forma di controllo della concorrenza multi-versione (MVCC) per supportare la sovrascrittura in modalità batch. Il passaggio da una versione all'altra dal lato utente non è percepibile, perché Druid lo gestisce caricando prima i nuovi dati e non appena i nuovi dati sono stati caricati, cambiando tutte le nuove query per utilizzare quelle nuovi segmenti. Quindi rilascia i vecchi segmenti pochi minuti dopo.

9.10.6 Ciclo di vita del segmento

Ogni segmento ha un ciclo di vita che coinvolge le seguenti tre aree principali:

- archivio dei metadati: contiene i metadati del segmento e vengono archiviati nell'archivio dei metadati una volta terminata la costruzione di un segmento.
- Archiviazione profonda: i file di dati del segmento vengono inseriti nell'archiviazione profonda una volta terminata la costruzione di un segmento. Ciò avviene immediatamente prima della pubblicazione dei metadati nell'archivio dei metadati.
- Disponibilità per l'interrogazione: i segmenti sono disponibili per l'interrogazione su alcuni server di dati Druid, come un'attività in tempo reale o un processo storico.

9.10.7 Injection dei dati

I principali metodi di ingestione di Druid sono tutti basati su pull e offrono garanzie transazionali (verrà caricato o tutto o niente).

- Metodi di ingestione "seekable-stream" supervisionati come Kafka e Kinesis . Con questi metodi, Druid esegue il commit degli offset del flusso nel proprio archivio di metadati insieme ai metadati del segmento, nella stessa transazione, in questo caso **Apache Druid** pubblica i segmenti prima che tutti i dati per un blocco temporale siano stati ricevuti;
- Inserimento batch basato su Hadoop; Ogni attività pubblica tutti i metadati del segmento in un'unica transazione.
- Importazione batch nativa . In modalità parallela, l'attività del supervisore pubblica tutti i metadati del segmento in un'unica transazione al termine delle attività secondarie.

9.11 Casi d'uso

9.11.1 Potenza di analisi dati in real-time e applicazioni dati

Un caso d'uso importante per **Apache Druid** è l'impiego di sistemi di acquisizione dati in tempo reale, query rapide e tempo di attività. **Druid** viene utilizzato per alimentare GUI per applicazioni analitiche o per **API** simultanee che necessitano di aggregazioni veloci. Funziona meglio su dati orientati agli eventi. Le aree di impiego più comuni:

- analisi dei flussi di clic
- analisi della telemetria di rete (monitoraggio delle prestazioni della rete)
- archiviazione delle metriche del server
- metriche delle prestazioni dell'applicazione
- operazioni **OLAP**

9.11.2 Attività e comportamento dell'utente

Apache Druid viene spesso utilizzato per effettuare misurazioni sul coinvolgimento degli utenti e il monitoraggio dei dati di test.

È utilizzato per calcolare metriche utente come conteggi distinti sia esattamente che approssimativamente.

9.11.3 Flussi di rete

Druid è comunemente usato per raccogliere e analizzare i flussi di rete. Druid viene utilizzato per suddividere arbitrariamente i dati del flusso lungo qualsiasi insieme di attributi. Questi attributi includono spesso attributi di base come IP e porta. **Druid** permette di gestire schemi flessibili di attributi.

9.11.4 Marketing digitale

Druid viene comunemente usato per archiviare e interrogare i dati pubblicitari online. Questi dati in genere provengono dagli ad server e sono fondamentali per misurare e comprendere le prestazioni delle campagne pubblicitarie, le percentuali di clic, i tassi di conversione (tassi di abbandono) e molto altro. In generale **Druid** è stato creato per gestire questo tipo di dati in grandi dimensioni.

Inoltre **Druid** può calcolare una serie di altre metriche per calcolare tendenze su grandi moli di dati.

9.11.5 Gestione delle prestazioni dell'applicazione

Druid viene spesso utilizzato per tenere traccia dei dati operativi generati dalle applicazioni. Similmente all'analisi dell'attività utente, in questo caso le metriche calcolate, o stimaste riguardano dati emessi dall'applicazione stessa.

9.11.6 Metriche IoT e dispositivi

Druid può essere utilizzato acquisire i dati generati dalla macchina in tempo reale ed eseguire rapide analisi ad hoc per misurare le prestazioni, ottimizzare le risorse hardware o identificare i problemi.

A differenza di altri database di serie temporali, **Druid** fornisce una serie di funzionalità di ricerca e partizionamento che gli permette di eseguire query molto velocemente.

9.11.7 Operazioni OLAP

Apache Druid viene utilizzato per accelerare l'esecuzione di query e potenziare le applicazioni. È progettato per un'elevata concorrenza e query in meno di un secondo, alimentando l'esplorazione interattiva dei dati attraverso un'interfaccia utente. Ciò fornisce una migliore soluzione interattiva con l'utente. Permette di effettuare analisi interattive su grandi moli di dati.

9.12 Modello dei dati

Druid memorizza i dati in *datasources* molto simili alle tabelle di un tradizionale database relazione e alle serie temporali. Il modello di dati di **Druid** presenta essenzialmente le seguenti componenti:

- **Timestamp principale:** Gli schemi Druid devono sempre includere un timestamp primario. Viene utilizzato per partizionare ed ordinare i dati. Inoltre il timestamp viene utilizzato per recuperare rapidamente i dati all'interno di un determinato intervallo di tempo.
- **Dimension:** sono colonne che Druid memorizza "così come sono". Si possono utilizzare per qualsiasi scopo: filtrare, applicare aggregatori e così via.
- **Metrics:** sono colonne che Druid archivia in forma aggregata e diventano più utili se attivi il rollup è una forma di aggregazione che comprime le dimensioni mentre aggrega i valori nelle metriche, ovvero comprime le righe ma conserva le informazioni di riepilogo.

9.12.1 Metrica

Le metriche sono colonne che **Druid** archivia in forma aggregata. Le metriche sono più utili quando abiliti il **rollup**.

Se specifichi una metrica, puoi applicare una funzione di aggregazione a ogni riga durante l'importazione. Il **rollup** è una forma di aggregazione che combina più righe con lo stesso valore di timestamp e gli stessi valori di dimensione; mentre aggrega i valori nelle metriche, ovvero comprime le righe ma conserva le informazioni di riepilogo.

9.12.2 Dimension

Le dimensioni sono colonne che Druid memorizza "così come sono". Le dimensioni possono essere utilizzate per qualsiasi scopo. Ad esempio per raggruppare, filtrare o applicare aggregatori alle dimensioni in fase di query, se necessario. Se il **rollup** è disabilitato le dimension vengono trattate come colonne da importare.

9.13 Schema design tips

- nell'importazione dei dati **Druid** con o senza **rollup**: aggregando parzialmente i tuoi dati durante l'importazione, riducendo potenzialmente il numero di righe, diminuendo l'impronta di archiviazione e migliorando le prestazioni delle query. Mentre senza **rollup** **Druid** archivia una riga per ogni riga di input.
- ogni riga in Druid deve avere un timestamp, tale informazione potrà essere utilizzata per successivi grappamenti.
- tutte le colonne del modello di archiviazione di **Druid** sono metriche ad eccezione del timestamp
- le colonne dimensione vengono archiviate così come sono
- le colonne delle metriche vengono archiviate preaggregate , quindi possono essere aggregate solo al momento della query.

9.13.1 Modello relazionale

In **Druid** in generale le *datasources* sono molto simili alle tabelle utilizzate nei modelli relazionali ma in questo caso si preferisce creare tabelle che permettano elaborazioni piatte: recupero delle informazioni senza l'utilizzo di join tra tabelle, ciò consente un aumento delle prestazioni.

9.13.2 Modello su serie storiche

Simile ai database delle serie temporali, il modello di dati di Druid richiede un timestamp. Druid non è un database di serie temporali, ma è una scelta naturale per l'archiviazione di dati di serie temporali. Il suo modello di dati flessibile consente di archiviare sia dati di serie temporali che non, anche nella stessa origine dati.

Per ottenere prestazioni ottimali di compressione e query in Druid per i dati delle serie temporali, è importante partizionare e ordinare in base al nome della metrica, come fanno spesso i database delle serie temporali.

9.13.3 Utilizzo degli Sketches

Nel momento in cui si vanno ad inserire dati **Druid** non li memorizza direttamente uno Sketch che poi può inserire nell'esecuzione delle query successive.

Ogni Sketch può essere acquisito dall'esterno di Druid o creato da dati grezzi al momento dell'acquisizione. È progettato per un solo particolare tipo di calcolo. In questo modo si riduce il **rollup** e si riduce la memoria utilizzata in fase di query.

9.13.4 Utilizzo del timestamp

Gli schemi **Druid** devono sempre includere un timestamp primario. Il timestamp principale viene utilizzato per partizionare e ordinare i dati, quindi dovrebbe essere il timestamp su cui filtrerai più spesso. Druid è in grado di identificare e recuperare rapidamente i dati corrispondenti agli intervalli di tempo della colonna timestamp primaria. Se lo schema lo richiede è possibile importare anche timestamp secondari.

9.13.5 Autorillevamento del tipo

È possibile fare in modo che **Druid** deduca lo schema e i tipi per i tuoi dati parzialmente o completamente impostando.

Nella deduzione del tipo **Druid** sceglie il tipo nativo più appropriato.

Per le colonne di tipo misto vengono archiviate nel tipo meno restrittivo che può rappresentare tutti i valori nella colonna.

9.14 Data rollup

Al momento dell'acquisizione dei dati **Apache Druid** può eseguire operazioni di raggruppamento sui dati per ridurre la quantità di dati grezzi da archiviare.

Da questo punto di vista il **rollup** non è altro che una forma di riepilogo e preaggregamento dei dati finalizzato a ridurre l'ordine di grandezza delle righe da archiviare.

Il **rollup** se da un lato porta una riduzione drastica degli eventi da analizzare, dall'altro non permette più di interrogare i singoli eventi.

Al momento dell'**injection** il **rollup** è abilitato di default: **Apache Druid** raggruppa in automatico gli eventi caratterizzati da stesso valore di **timestamp** e stesso valore per le **Dimension**.

Se il **rollup** è disabilitato, **Apache Druid** caricherà gli eventi così come sono stati generati, senza alcun riepilogo.

9.14.1 Casi d'uso

È conveniente andare ad utilizzare il **rollup** se si desiderano prestazioni ottimali e vicoli di spazio rigorosi o quando non si ha necessità di avere a disposizione dati grezzi con alta cardinalità.

Mentre se si ha bisogno di risultati per righe, l'utilizzo del **rollup** non è la scelta più adatta; è necessario in quel caso effettuare delle operazioni di **GROUP BY** a query time.

In ogni caso, per soddisfare entrambi i casi d'uso appena citati è possibile andare a configurare 2 datasource: uno con **rollup** attivo, con il riepilogo degli eventi generati, e uno con **rollup** disabilitato contenente tutti i dati grezzi.

9.14.2 Tipi di rollup

A seconda del metodo di importazione, **Apache Druid** presenta due metodi differenti di **rollup**.

- rollup perfetto: **Apache Druid** aggrega perfettamente i dati a injection time;
- Best-effort rollup: **Apache Druid** potrebbe non riuscire ad aggregare perfettamente di input pertanto si potrebbe verificare che più segmenti dati contengano dati con stesso timestamp e stesso valore di Dimension.

In generale I metodi di inserimento che garantiscono un rollup perfetto utilizzano un ulteriore passaggio di pre-elaborazione per determinare gli intervalli e il partizionamento prima dell'inserimento dei dati. Questa fase di pre-elaborazione esegue la scansione dell'intero set di dati di input. Anche se questo passaggio aumenta il tempo necessario per l'inserimento, fornisce le informazioni necessarie per un rollup perfetto.

Mentre i metodi che garantiscono il **Best-effort rollup** lo fanno per i seguenti motivi:

- la parallelizzazione dell'inserimento avviene senza un passaggio di mescolamento necessario per il **rollup** perfetto;
- Il metodo di importazione utilizza la pubblicazione incrementale , il che significa che finalizza e pubblica i segmenti prima che tutti i dati per un blocco temporale siano stati ricevuti, tutti i tipi di importazione di streaming vengono eseguiti in questa modalità.

9.15 Le lookups

In **Apache Druid** le **lookup** sono una funzionalità che consentono di sostituire i valori delle **Dimension** con nuovi valori. Tale concetto può essere paragonato alla unione di tabelle in un **data warehouse**.

Le tabelle di **lookup** sono costituite da un campo **chiave** a cui viene associato un campo **valore** che andrà a sostituire la chiave.

Le tabelle di **lookup** non hanno cronologia e lavorano indipendentemente dall'intervallo di tempo. su cui si esegue la query: restituiscono sempre il dato corrente.

9.15.1 Sintassi SQL di query con l'utilizzo delle tabelle di lookup

In **Apache Druid** le tabelle di **lookup** possono essere utilizzate attraverso funzione *LOOKUP(keys, lookup_table_name)*.

```
SELECT LOOKUP(store, 'store_to_country') AS country, SUM(revenue)
FROM sales
GROUP BY 1
```

Le tabelle di **lookup** consentendo le operazione di tipo join.

```
SELECT store_to_country.v AS country, SUM(sales.revenue) AS country_revenue
FROM sales INNER JOIN lookup.store_to_country ON sales.store = store_to_country.k
GROUP BY 1
```

9.15.2 Tipi di lookup

In generale le tabelle di **lookup** possono essere di due tipi: iniettive e non iniettive.

Nel primo tipo rientrano tutti i casi d'uso legati alla mappatura univoca tra chiave e valore; mentre nel secondo caso rientrano tutte le mappature nelle quali più chiavi riferiscono lo stesso valore.

Nel momento in cui si creano tabelle di **lookup** iniettive **Apache Druid** è in grado di applicare ulteriori ottimizzazioni nella esecuzione delle query.

Tale fatto deve essere tenuto in considerazione nel caso di aggregazioni, in quanto, nel caso in cui chiavi diverse puntino allo stesso valore (tabella di **lookup** non iniettiva), nel risultato di una query compariranno due valori che avrebbero dovuto essere aggregati.

9.15.3 Casi d'uso

Le tabelle di **lookup** sono particolarmente utili nei processi di **Data enrichment**.

Inoltre possono andare a risolvere anche essere una possibile soluzione al problema degli aggiornamenti. In questo caso ogni aggiornamento di stato viene visto come un nuovo evento e viene aggiunto al **datasource** originale.

Per estrarre sempre il dato più recente si utilizza una **lookup** iniettiva che mappa ogni chiave con il valore dell'attributo richiesto. Essendo le tabelle di **lookup** indipendenti dagli intervalli temporali, verrà reatsituito il valore più recente.

9.16 Confronto Apache Druid e PostgreSQL

Per andare ad evidenziare pregi e difetti di **Apache Druid** è stato scelto di confrontarlo con un database relazionale come **PostgreSQL**, nella esecuzione di query su un dataset di 1 milione di righe.

Il test effettuato non usufruisce della cache.

Lo schema logico del dataset è il seguente:

```
CREATE TABLE transazione
(
  __time timestamp,
  cliente text,
  prodotto text,
  quantita int,
  gradimento int
)
```

9.16.1 Query

```
query1:
SELECT * FROM transazione
query2:
SELECT DATA_TRUNC('MINUTE',__time), cliente, prodotto, COUNT(*)
FROM transazione
WHERE gradimento>3
GROUP BY DATA_TRUNC('MINUTE',__time), cliente, prodotto
```

```

query3:
SELECT DATA_TRUNC('HOUR',__time), cliente, prodotto, gradimento
FROM transazione
WHERE gradimento>3
GROUP BY DATA_TRUNC('HOUR',__time), cliente, prodotto, gradimento

```

	Apache Druid [s]	PostgreSQL [s]
query1	0.007	0.77
query2	0.45	0.57
query3	0.30	0.60

10 Kubernetes

10.1 Panoramica e caratteristiche principali

Kubernetes è una piattaforma open source utilizzata per l'orchestrazione e la gestione dei nodi.

Il nome Kubernetes deriva dal greco, significa timoniere o pilota.

È un servizio estendibile per la gestione di carichi di lavoro e servizi containerizzati. Presenta un grande ecosistema in rapida crescita.

10.2 Casi d'uso

I container sono un buon modo per raggruppare ed eseguire le applicazioni. In un ambiente di produzione, è necessario gestire i container ed assicurarsi che vi siano tempi di inattività. In tali casi entra in gioco **Kubernetes** che fornisce un **framework** finalizzato alla esecuzione di sistemi distribuiti in modo resiliente.

10.3 Funzionalità

Kubernetes fornisce le seguenti funzionalità:

- **rilevamento dei servizi e bilanciamento del carico:** **Kubernetes** è in grado di bilanciare il carico e distribuire il traffico di rete in modo che la distribuzione sia stabile;
- **orchestrazione dello storage:** **Kubernetes** consente di montare storage locali, provider di cloud pubblico e altro;
- **rollout e rollback automatizzati:** puoi descrivere lo stato desiderato per i tuoi container distribuiti utilizzando Kubernetes e può cambiare lo stato effettivo nello stato desiderato a una velocità controllata.
- **bin packing automatico:** è possibile impartire a **Kubernetes** quanta CPU e memoria (RAM) ha bisogno ogni container; inoltre **Kubernetes** può adattare i container ai tuoi nodi per utilizzare al meglio le tue risorse;

- **riparazione automatica:** **Kubernetes** è in grado di rilevare i container che hanno avuto esito negativo, sostituirli e terminare i container non rispondono allo stato definito dall'utente;
- **gestione dei secrets e della configurazione:** **Kubernetes** ti consente di archiviare e gestire informazioni riservate

10.4 Architettura

Kubernetes viene distribuito in un cluster.

Un cluster è costituito da un insieme di macchine **worker**, denominate nodi, che eseguono le applicazioni containerizzate. Ogni cluster ha almeno un nodo **worker**.

I nodi **worker** ospitano i **POD**, che rappresentano le minime unità di calcolo, distribuibili e gestibili su **Kubernetes**. Il **control pane** gestisce i nodi di lavoro e i **POD** del cluster. In ambienti di produzione per garantire tolleranza ai guasti e alta disponibilità il **control pane** e **POD** sono eseguiti su più calcolatori e un cluster esegue i nodi.

10.4.1 I componenti del control pane

I componenti di **control pane** prendono decisioni globali sul cluster, oltre a rispondere ad eventi del cluster. Per semplicità i componenti di **control pane** vengono tutti eseguiti sulla stessa macchina.

kube-apiserver È un componente del **control pane** di **Kubernetes** che espone l'**API Kubernetes**. L'implementazione principale è un **kube-apiserver**, progettato per scalare orizzontalmente, distribuendosi su più istanze. È possibile andare a bilanciare il traffico tra tali istanze.

ecc È l'archivio di valore chiave coerente utilizzato come archivio di backup di **Kubernetes** per tutti i dati del cluster.

kube-scheduler È un componente del **control pane** a cui è affidato il compito di controllare la creazione dei **POD** senza assegnare nodi su cui eseguirli.

Per prendere le decisioni di pianificazione vengono presi in considerazione i seguenti fattori:

- requisiti di risorse individuali e collettivi;
- vincoli hardware/software/policy;
- specifiche di affinità e anti-affinità;
- località dei dati;
- interferenza tra carichi di lavoro e scadenze.

kube-controller-manager È il componente adibito al controllo dei processi.

Logicamente ogni **controller** è un processo separato, ma per ridurre la complessità, sono tutti compilati in un unico binario ed eseguiti in un unico processo.

cloud-controller-manager È il componente che incorpora la logica di controllo specifica del cloud. Il **cloud-controller-manager** consente di collegare il cluster con l'**API** del provider cloud; separa i componenti che interagiscono con quella piattaforma cloud dai componenti che interagiscono solo con il tuo cluster. Se esegui **Kubernetes** nei tuoi locali, il cluster non dispone di un **cloud-controller-manager**.

10.4.2 I componenti del nodo

I componenti del nodo vengono eseguiti su ogni nodo, mantenendo i pod in esecuzione e fornendo l'ambiente di runtime **Kubernetes**.

kubelet È un agente che viene eseguito per ogni nodo del cluster e si assicura che i container stiamo eseguendo un **POD**.

kube-proxy È un proxy di rete che viene eseguito su ogni nodo del cluster, implementando parte del servizio **Kubernetes**. Mantiene le regole di rete sui nodi. Queste regole di rete consentono la comunicazione di rete ai **POD** dalle sessioni di rete all'interno o all'esterno del tuo cluster.

10.5 Il funzionamento

Kubernetes esegue i compiti assegnati di lavoro inserendo i container nei **POD** per l'esecuzione sui nodi. Un nodo può essere una macchina virtuale o fisica, a seconda del cluster.

Ogni nodo è gestito da un **control pane** e contiene i servizi necessari per l'esecuzione di un **POD**. I componenti su un nodo includono il **kubelet**, un runtime del container, e il **kube-proxy**.

Esistono due modi principali per aggiungere i nodi: **Server API**

- il kubelet su un nodo si registra automaticamente nel piano di controllo;
- viene aggiunto manualmente un oggetto di tipo **Node**.

Una volta effettuata la creazione dell'oggetto Nodo o il **kubelet** su un nodo, il **control pane** che il nuovo oggetto di tipo **Node** sia integro (ovvero tutti i servizi necessari sono in esecuzione, idoneo ad eseguire un POD).

Un esempio di configurazione di un nodo, in formato Json può essere il seguente:

```
{
  "kind": "Node",
  "apiVersion": "v1",
  "metadata": {
    "name": "10.240.79.157",
    "labels": {
      "name": "my-first-k8s-node"
    }
  }
}
```


10.5.1 Unicità del nodo

Il nome identifica in modo univoco un nodo. Due nodi non possono avere lo stesso nome, perchè **Kubernetes** presuppone che due risorse con lo stesso nome rappresentino la stessa risorsa. Se tale condizione non venisse rispettata si verificherebbero delle incoerenze di stato dei nodi.

Se il nodo deve essere sostituito o aggiornato in modo significativo, l'oggetto nodo esistente deve essere prima rimosso dal server API e aggiunto nuovamente dopo l'aggiornamento.

10.5.2 Lo stato di un nodo

Lo stato di un nodo contiene le seguenti informazioni:

- Indirizzi: contiene le informazioni:
 - **HostName**: il nome host riportato dal kernel del nodo;
 - **ExternalIP**: in genere l'indirizzo IP del nodo che è instradabile esternamente;
 - **InternalIP**: in genere l'indirizzo IP del nodo che è instradabile solo all'interno del cluster.
 - le conditions: campo che descrive lo stato di tutti i nodi attivi;
 - * **Ready**: True se il nodo è integro e pronto ad accettare pod, False se il nodo non è integro e non accetta pod e Unknown se il controller del nodo non ha ricevuto notizie dal nodo nell'ultimi 40 secondi;
 - * **DiskPressure**: True se la capacità del disco è bassa;
 - * **MemoryPressure**: True se la memoria del nodo è bassa;
 - * **PIDPressure**: True se ci sono troppi processi sul nodo;
 - * **NetworkUnavailable**: True se la rete per il nodo non è configurata correttamente.

10.6 Heartbeats

Gli **Heartbeats** o battiti del cuore aiutano il tuo cluster a determinare la disponibilità di ciascun nodo e ad agire quando vengono rilevati errori.

Per i nodi ci sono 2 forme di battiti del cuore:

- aggiornamenti al *.status* di un nodo;
- allocare oggetti all'interno del *kube-node-lease* spazio dei nomi; ogni nodo ha un oggetto **Lease** associato;

Rispetto agli aggiornamenti *.status* di un nodo, un lease è una risorsa leggera. L'utilizzo dei lease per gli heartbeat riduce l'impatto sulle prestazioni di questi aggiornamenti per i cluster di grandi dimensioni.

Il **kubelet** è responsabile della creazione e dell'aggiornamento dei *.statusnodi* e dell'aggiornamento dei relativi *lease*.

In particolare si occupa di:

- aggiorna il nodo *.status* quando c'è un cambiamento di stato o se non c'è stato alcun aggiornamento per un intervallo configurato (l'intervallo prededefinito è di 5 minuti);
- Il **kubelet** crea e quindi aggiorna il suo oggetto **Lease** ogni 10 secondi; gli aggiornamenti del **Lease** si verificano indipendentemente dagli aggiornamenti del **Lease**.

10.7 Il controllore del nodo

È un componente del **control pane** che gestisce i vari aspetti del nodo.

I compiti che svolge il controllore sono:

- assegnare un blocco CIDR al nodo quando è registrato;
- mantenere aggiornato l'elenco interno dei nodi del controller del nodo con l'elenco delle macchine disponibili del fornitore di servizi cloud;
- monitora la salute dei nodi.

Per impostazione predefinita, il controller del nodo controlla lo stato di ciascun nodo ogni 5 secondi.

10.8 L'elezione del capo

Kubernetes utilizza anche i **lease** per garantire che solo un'istanza di un componente sia in esecuzione in un dato momento.

10.9 Garbage Collection

Con **Garbage Collection** si va ad intendere tutti i meccanismi che **Kubernetes** utilizza per ripulire le risorse del cluster.

Molti oggetti di **Kubernetes** utilizzano l'**owner references**, che indicano al **control pane** quali oggetti dipendono da altri. **Kubernetes** elimina tutte le risorse correlate ad un oggetto prima di eliminare quest'ultimo.

10.10 Foreground cascading deletion

Kubernetes controlla ed elimina gli oggetti che non hanno più riferimenti al proprietario. Quando elimini un oggetto, puoi controllare se **Kubernetes** elimina automaticamente i dipendenti dell'oggetto, in un processo chiamato eliminazione a cascata.

Esistono due tipi di eliminazione a cascata:

- Eliminazione a cascata in primo piano: l'oggetto in questione entra in una fase di "eliminazione in corso", L'oggetto rimane visibile tramite l'API **Kubernetes** fino al completamento del processo di eliminazione;
- Eliminazione a cascata in background: il server **API Kubernetes** elimina immediatamente l'oggetto proprietario e il controller pulisce gli oggetti dipendenti in background. Quando **Kubernetes** elimina un oggetto proprietario, i dipendenti rimasti vengono chiamati oggetti orfani.

10.11 I service

In **Kubernetes**, un servizio è un metodo per esporre un'applicazione di rete in esecuzione come una o più **PODS**.

10.12 Il ConfigMap

Un **ConfigMap** è un oggetto API utilizzato per archiviare dati, possono essere utilizzate come variabili d'ambiente.

10.13 Configurazione con minikube

In questa sezione viene descritto come eseguire un'app d'esempio su **Kubernetes**.

Tale esempio presuppone l'esecuzione su singola macchina del cluster e l'utilizzo di **minikube**.

Gli obiettivi della seguente demo di esempio sono:

- distribuire un'applicazione di esempio su **minikube**;
- eseguire l'applicazione;
- visualizzare i registri dell'applicazione.

10.13.1 Svolgimento

Prima di tutto assicurarsi di aver configurato **minikube** sulla propria macchina.

L'installazione può essere effettuata attraverso i seguenti comandi:

```
|| curl -LO  
||     https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64  
|| sudo install minikube-linux-amd64 /usr/local/bin/minikube
```

1. Creare un cluster **minikube**

```
|| minikube start
```

È possibile abilitare la dashboard dove eseguire operazioni di deployment o creare risorse **Kubernetes** con il comando

```
|| minikube dashboard
```


2. Un **POD** rappresenta un gruppo di container, collegati per lo svolgimento di un determinato compito

Con il comando **kubectl create** è possibile creare un deployment che gestisce tale **POD**. Ed il **POD** esegue un container sulla immagine fornita

```
|| kubectl create deployment hello-node  
||     --image=registry.k8s.io/e2e-test-images/agnhost:2.39 -- /agnhost  
||     netexec --http-port=8080
```

3. Per visualizzare la distribuzione dei nodi, utilizzare il comando

```
|| kubectl get deployments
```




images/output_get_deployment.png

Figure 3: get deployment

4. Per visualizzare il POD utilizzare il comando

```
||      kubectl get pods
```



images/output_pods.png

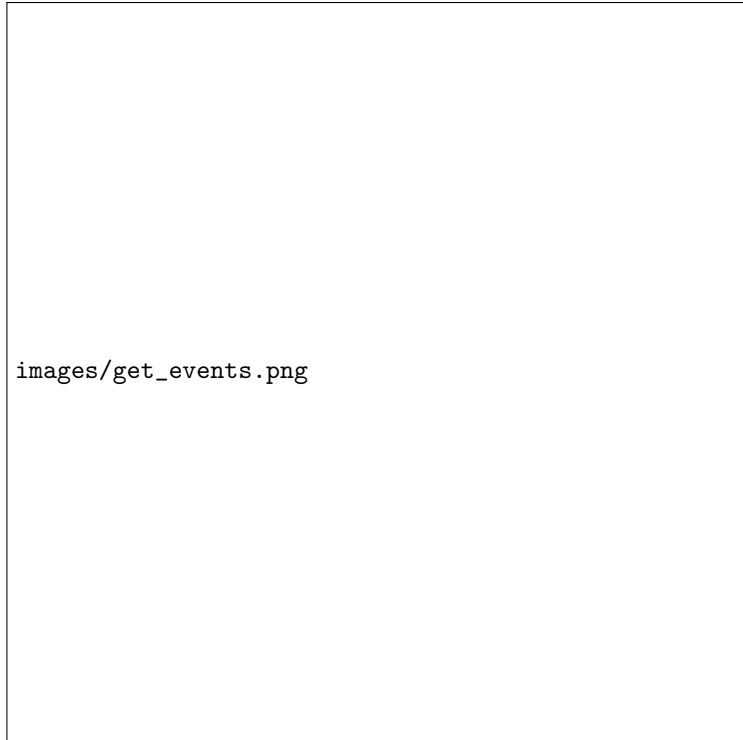
Figure 4: Output PODS

5. Per visualizzare gli eventi del cluster

```
||      kubectl get events
```

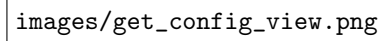
6. Per visualizzare le configurazioni kubectl

```
||      kubectl config view
```

A rectangular box containing the text 'images/get_events.png' in a monospaced font, serving as a placeholder for a screenshot of the 'Get events' interface.

images/get_events.png

Figure 5: Get events

A rectangular box containing the text 'images/get_config_view.png' in a monospaced font, serving as a placeholder for a screenshot of the 'Config view' interface.

images/get_config_view.png

Figure 6: Config view

Assegnare un **PODS** a un nodo.

1. Creare un POD;

```
apiVersion: v1
kind: Pod
metadata:
  name: nome-del-pod
  labels:
    app: tua-app
spec:
  containers:
  - name: nome-del-contenitore
    image: nome-immagine-docker:tag
    ports:
    - containerPort: porta-da-esporre
```

Ed eseguire il comando

```
kubectl apply -f nome-file-yaml.yaml
```

2. elencare i nodi del cluster;

```
kubectl get nodes --show-labels
```

3. scegliere un nodo;

```
kubectl label nodes <label_del_nodo>
```

Ed eseguire il comando

```
kubectl apply -f nome-file-yaml.yaml
```

10.14 Configurazione attraverso file kubeconfig

Il file **kubeconfig** viene utilizzato per immagazzinare le informazioni di autenticazione per **kubectl**. La posizione predefinita del **kubeconfig** file è */.kube*.

In generale invece di utilizzare il nome **kubeconfig**, il file viene semplicemente denominato **config**.

10.14.1 Composizione

il **kubeconfig** file è un file YAML contenente tutti i gruppi di **cluster**, di **utenti** e di **contesti**.

- Un **cluster** è un cluster **Kubernetes**;
- un utente è una credenziale utilizzata per interagire con **API Kubernetes**;
- un **contesto** è una combinazione di un cluster e un utente, ogni volta che si esegue il comando **kubectl** si fa riferimento ad un contesto presente all'interno del file **kubeconfig**.

10.15 Esempio di deployment di minikube

```
apiVersion: v1
kind: Config
clusters:
- name: minikube
  cluster:
    certificate-authority: /home/hector/.minikube/ca.crt
    server: https://192.168.39.217:8443
users:
- name: minikube
  user:
    client-certificate: /home/hector/.minikube/profiles/minikube/client.crt
    client-key: /home/hector/.minikube/profiles/minikube/client.key
contexts:
- name: minikube
  context:
    cluster: minikube
    namespace: default
    user: minikube
current-context: minikube
```

- La sezione cluster elenca tutti i cluster che sono già stati connessi. In questo caso, ce solo minikube (il nome è arbitrario). All'interno del **cluster** sono presenti due chiavi:
 - **certificate-authority**: contiene il certificato per l'autorità di certificazione. Può trattarsi di un certificato di un percorso file o di una stringa Base64 del formato PEM (Privacy Enhanced Mail) del certificato;
 - **server**: è l'indirizzo del server;
- la sezione utenti elenca tutti gli utenti già utilizzati per connettersi a un cluster. Esistono alcune possibili chiavi per un utente:
 - **client-certificate**: contiene un certificato per l'utente firmato dalla CA Kubernetes. È un percorso file o di una stringa Base64 nel formato PEM del certificato;
 - **chiave-client**: contiene la chiave che ha firmato il certificato client;
 - **token**: contiene un token per questo utente quando non è presente alcun certificato;
- La sezione contexts specifica una combinazione di un utente e un cluster. Inoltre definisce uno spazio dei nomi predefinito per questa coppia. Il nome del contesto è arbitrario, ma l'utente e il cluster devono essere predefiniti all'interno del **kubeconfig** file.

10.15.1 Configurazione l'accesso a più cluster

Per definire l'accesso a più cluster, utilizzando il file di configurazione, procedere nel seguente modo.

Per passare da un cluster ad un altro utilizzare il comando:

```
|| kubectl config use-context
```

Supponiamo di avere due cluster, uno per il lavoro di sviluppo e uno per il lavoro di test. I tuoi sviluppatori front-end lavorano in uno spazio dei nomi chiamato frontend i tuoi sviluppatori di storage lavorano in uno spazio dei nomi chiamato storage. Nel tuo testcluster, gli sviluppatori lavorano nello spazio dei nomi predefinito o creano spazi dei nomi ausiliari come meglio credono. L'accesso al cluster di sviluppo richiede l'autenticazione tramite certificato. L'accesso al cluster di test richiede l'autenticazione tramite nome utente e password. Passi da seguire:

1. Crea una directory denominata **config-exercise**, con all'interno un file denominato **config-demo** con il seguente contenuto

```
|| apiVersion: v1
|| kind: Config
|| preferences: {}
||
|| clusters:
|| - cluster:
||   name: development
|| - cluster:
||   name: test
||
|| users:
|| - name: developer
|| - name: experimenter
||
|| contexts:
|| - context:
||   name: dev-frontend
|| - context:
||   name: dev-storage
|| - context:
||   name: exp-test
```

2. aggiungere i dettagli del cluster

```
|| kubectl config --kubeconfig=config-demo set-cluster development
||   --server=https://1.2.3.4 --certificate-authority=fake-ca-file
|| kubectl config --kubeconfig=config-demo set-cluster test
||   --server=https://5.6.7.8 --insecure-skip-tls-verify
```

3. aggiungere i dettagli degli utenti

```
|| kubectl config --kubeconfig=config-demo set-credentials developer
||   --client-certificate=fake-cert-file --client-key=fake-key-seefile
|| kubectl config --kubeconfig=config-demo set-credentials experimenter
||   --username=exp --password=some-password
```

4. aggiungere i dettagli del contesto di configurazione


```
kubectl config --kubeconfig=config-demo set-context dev-frontend
--cluster=development --namespace=frontend --user=developer
kubectl config --kubeconfig=config-demo set-context dev-storage
--cluster=development --namespace=storage --user=developer
kubectl config --kubeconfig=config-demo set-context exp-test --cluster=test
--namespace=default --user=experimenter
```

5. per visualizzare il risultato finale

```
kubectl config --kubeconfig=config-demo view
```

```
apiVersion: v1
clusters:
- cluster:
  certificate-authority: fake-ca-file
  server: https://1.2.3.4
  name: development
- cluster:
  insecure-skip-tls-verify: true
  server: https://5.6.7.8
  name: test
contexts:
- context:
  cluster: development
  namespace: frontend
  user: developer
  name: dev-frontend
- context:
  cluster: development
  namespace: storage
  user: developer
  name: dev-storage
- context:
  cluster: test
  namespace: default
  user: experimenter
  name: exp-test
current-context: ""
kind: Config
preferences: {}
users:
- name: developer
  user:
    client-certificate: fake-cert-file
    client-key: fake-key-file
- name: experimenter
  user:
    # Documentation note (this comment is NOT part of the command output).
    # Storing passwords in Kubernetes client config is risky.
```

```

# A better alternative would be to use a credential plugin
# and store the credentials separately.
# See
  https://kubernetes.io/docs/reference/access-authn-authz/authentication/#client-go-credential-plugins
password: some-password
username: exp

```

Per applicare tale configurazione utilizzare il comando:

```
kubectl -f nome_file.yml
```

10.16 Esempio di deployments con Apache Druid

1. Creazione del service **Zookeeper**

```

apiVersion: v1
kind: Service
metadata:
  name: zk-service
spec:
  selector:
    app: zookeeper
  ports:
    - name: client
      port: 2181
      targetPort: 2181

```

2. definire la **ConfigMap** per **Apache Druid**

```

apiVersion: v1
kind: ConfigMap
metadata:
  name: druid-config
data:
  druid.common.runtime.properties: |-
    druid.server.type=historical
    druid.zk.service.host=zk-service:2181
    # Altre impostazioni specifiche per il nodo historical

  druid-broker.runtime.properties: |-
    druid.server.type=broker
    druid.zk.service.host=zk-service:2181
    # Altre impostazioni specifiche per il nodo broker

  druid-coordinator.runtime.properties: |-
    druid.server.type=coordinator
    druid.zk.service.host=zk-service:2181
    # Altre impostazioni specifiche per il nodo coordinator

```

3. definizione dei nodi Historical

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: druid-historical
spec:
  replicas: 1
  selector:
    matchLabels:
      app: druid-historical
  template:
    metadata:
      labels:
        app: druid-historical
    spec:
      containers:
        - name: druid
          image: your-druid-image:latest
          ports:
            - containerPort: 8083 # Porta per il nodo historical
          volumeMounts:
            - name: druid-config-volume
              mountPath: /opt/druid/conf/druid
      volumes:
        - name: druid-config-volume
          configMap:
            name: druid-config
            items:
              - key: druid.common.runtime.properties
                path: runtime.properties

```

4. definizione del nodo **Broker**

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: druid-broker
spec:
  replicas: 1
  selector:
    matchLabels:
      app: druid-broker
  template:
    metadata:
      labels:
        app: druid-broker
    spec:
      containers:
        - name: druid
          image: your-druid-image:latest
          ports:

```

```

        - containerPort: 8082 # Porta per il nodo broker
    volumeMounts:
        - name: druid-config-volume
          mountPath: /opt/druid/conf/druid
    volumes:
        - name: druid-config-volume
          configMap:
            name: druid-config
            items:
                - key: druid-broker.runtime.properties
                  path: runtime.properties

```

5. definizione del nodo **Coordinator**

```

apiVersion: apps/v1
kind: Deployment
metadata:
    name: druid-coordinator
spec:
    replicas: 1
    selector:
        matchLabels:
            app: druid-coordinator
    template:
        metadata:
            labels:
                app: druid-coordinator
        spec:
            containers:
                - name: druid
                  image: your-druid-image:latest
                  ports:
                      - containerPort: 8081 # Porta per il nodo coordinator
                  volumeMounts:
                      - name: druid-config-volume
                        mountPath: /opt/druid/conf/druid
            volumes:
                - name: druid-config-volume
                  configMap:
                    name: druid-config
                    items:
                        - key: druid-coordinator.runtime.properties
                          path: runtime.properties

```

6. definizione del service **Druid**

```

apiVersion: v1
kind: Service
metadata:
    name: druid-service

```

```
spec:
  ports:
    - name: druid-historical
      port: 8083
      targetPort: 8083
      selector:
        app: druid-historical
    - name: druid-broker
      port: 8082
      targetPort: 8082
      selector:
        app: druid-broker
    - name: druid-coordinator
      port: 8081
      targetPort: 8081
      selector:
        app: druid-coordinator
```

7. definire il nodo **Postgres**

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: postgres
spec:
  replicas: 1
  selector:
    matchLabels:
      app: postgres
  template:
    metadata:
      labels:
        app: postgres
    spec:
      containers:
        - name: postgres
          image: postgres:latest
          ports:
            - containerPort: 5432 # Porta predefinita di PostgreSQL
          env:
            - name: POSTGRES_USER
              value: your_username
            - name: POSTGRES_PASSWORD
              value: your_password
            - name: POSTGRES_DB
              value: your_database_name
          volumeMounts:
            - name: postgres-data
              mountPath: /var/lib/postgresql/data
      volumes:
```

```

- name: postgres-data
  emptyDir: {}

```

8. definire il service **Postgres**

```

apiVersion: v1
kind: Service
metadata:
  name: postgres
spec:
  selector:
    app: postgres
  ports:
    - name: postgres
      port: 5432
      targetPort: 5432

```

9. dopo aver configurato i precedenti file **YAML** eseguire i seguenti comandi (i nomi dei file sono arbitrari)

```

kubectl apply -f druid-configmap.yml
kubectl apply -f druid-historical.yml
kubectl apply -f druid-broker.yml
kubectl apply -f druid-coordinator.yml
kubectl apply -f druid-service.yml
kubectl apply -f postgres-config.yml
kubectl apply -f postgres-service.yml

```

10. per verificare lo stato del cluster, eseguire i seguenti comandi

```

kubectl get pods      # Visualizza lo stato dei pod
kubectl get services  # Visualizza lo stato dei service
kubectl get configmaps # Visualizza lo stato dei ConfigMap

```

11 Implementazione

11.1 Apache Kafka

Le note riportate si riferiscono ad un sistema operativo Linux dove è già stata scaricata una versione di **Apache Kafka**. I comandi devono essere eseguiti dalla cartella di installazione.

- mandare in esecuzione il gestore **Zookeeper**

```

bin/zookeeper-server-start.sh config/zookeeper.properties

```

- mandare in esecuzione il **Broker**

```

bin/kafka-server-start.sh config/server.properties

```

- creare un nuovo **topic** name

```
|| bin/kafka-topics.sh--create --topic topic--name--bootstrap-serverIP:port  
|| server
```

- creare un **produttore**

```
|| bin/kafka-console-producer.sh --topic topic--name --bootstrap-server  
|| IP:port server
```

- creare un **consumatore**

```
|| bin/kafka-console-consumer.sh--topic topic--name--from-beginning  
|| --bootstrap-server IP:port server
```

11.2 Creazione di un generatore e consumatore di eventi in python con Apache Kafka

Come prima operazione di è andati a creare un producer di eventi in python che simuli il cambiamento di stato di un valore.

```
from time import sleep
from json import dumps
from kafka import KafkaProducer
import random
import datetime
import json

producer = KafkaProducer(
    bootstrap_servers = ['localhost:9092', 'localhost:9093', 'localhost:9094'],
    value_serializer = lambda x: json.dumps(x).encode('utf-8')
)
print("Connect succesfully")

for n in range(50):
    for j in range(10):
        my_data = {"timestamp": str(datetime.datetime.now()), "id" :
                    str(random.randint(0,999)), "value": random.randint(0,1)}
        producer.send('druid', value = my_data)
        print("Send")
        sleep(1)
    print("End")
```

11.2.1 Svolgimento

- Come prima operazione abbiamo mandato in esecuzione il kluster di nodi **Apache Kafka** formato da 1 nodo **Zookeeper** e 3 nodi **Broker** (Kafka1, Kafka2, Kafka3).

```
networks:
  kafka-druid:
    name: kafka-druid
    driver: bridge
    external: true

services:
  zoo1:
    image: confluentinc/cp-zookeeper:7.3.2
    hostname: zoo1
    networks:
      - kafka-druid
    container_name: zoo1
    ports:
      - "2182:2181"
    environment:
```



```
ZOOKEEPER_CLIENT_PORT: 2181
ZOOKEEPER_SERVER_ID: 1
ZOOKEEPER_SERVERS: zoo1:2888:3888
```

kafka1:

```
image: confluentinc/cp-kafka:7.3.2
```

```
hostname: kafka1
```

```
networks:
```

```
- kafka-druid
```

```
container_name: kafka1
```

```
ports:
```

```
- "9092:9092"
```

```
- "29092:29092"
```

```
environment:
```

```
KAFKA_ADVERTISED_LISTENERS:
```

```
INTERNAL://kafka1:19092,EXTERNAL://${DOCKER_HOST_IP:-127.0.0.1}:9092,DOCKER://host.docker.intern
```

```
KAFKA_LISTENER_SECURITY_PROTOCOL_MAP:
```

```
INTERNAL:PLAINTEXT,EXTERNAL:PLAINTEXT,DOCKER:PLAINTEXT
```

```
KAFKA_INTER_BROKER_LISTENER_NAME: INTERNAL
```

```
KAFKA_ZOOKEEPER_CONNECT: "zoo1:2181"
```

```
KAFKA_BROKER_ID: 1
```

```
KAFKA_LOG4J_LOGGERS:
```

```
"kafka.controller=INFO,kafka.producer.async.DefaultEventHandler=INFO,state.change.logger=INFO"
```

```
KAFKA_AUTHORIZER_CLASS_NAME: kafka.security.authorizer.AclAuthorizer
```

```
KAFKA_ALLOW_EVERYONE_IF_NO_ACL_FOUND: "true"
```

```
depends_on:
```

```
- zoo1
```

kafka2:

```
image: confluentinc/cp-kafka:7.3.2
```

```
hostname: kafka2
```

```
networks:
```

```
- kafka-druid
```

```
container_name: kafka2
```

```
ports:
```

```
- "9093:9093"
```

```
- "29093:29093"
```

```
environment:
```

```
KAFKA_ADVERTISED_LISTENERS:
```

```
INTERNAL://kafka2:19093,EXTERNAL://${DOCKER_HOST_IP:-127.0.0.1}:9093,DOCKER://host.docker.intern
```

```
KAFKA_LISTENER_SECURITY_PROTOCOL_MAP:
```

```
INTERNAL:PLAINTEXT,EXTERNAL:PLAINTEXT,DOCKER:PLAINTEXT
```

```
KAFKA_INTER_BROKER_LISTENER_NAME: INTERNAL
```

```
KAFKA_ZOOKEEPER_CONNECT: "zoo1:2181"
```

```
KAFKA_BROKER_ID: 2
```

```
KAFKA_LOG4J_LOGGERS:
```

```
"kafka.controller=INFO,kafka.producer.async.DefaultEventHandler=INFO,state.change.logger=INFO"
```

```
KAFKA_AUTHORIZER_CLASS_NAME: kafka.security.authorizer.AclAuthorizer
```

```

    KAFKA_ALLOW_EVERYONE_IF_NO_ACL_FOUND: "true"
depends_on:
  - zoo1

kafka3:
  image: confluentinc/cp-kafka:7.3.2
  hostname: kafka3
  networks:
    - kafka-druid
  container_name: kafka3
  ports:
    - "9094:9094"
    - "29094:29094"
  environment:
    KAFKA_ADVERTISED_LISTENERS:
      INTERNAL://kafka3:19094,EXTERNAL://${DOCKER_HOST_IP:-127.0.0.1}:9094,
      DOCKER://host.docker.internal:29094
    KAFKA_LISTENER_SECURITY_PROTOCOL_MAP:
      INTERNAL:PLAINTEXT,EXTERNAL:PLAINTEXT,DOCKER:PLAINTEXT
    KAFKA_INTER_BROKER_LISTENER_NAME: INTERNAL
    KAFKA_ZOOKEEPER_CONNECT: "zoo1:2181"
    KAFKA_BROKER_ID: 3
    KAFKA_LOG4J_LOGGERS:
      "kafka.controller=INFO,kafka.producer.async.DefaultEventHandler=INFO,state.change.logger=INFO"
    KAFKA_AUTHORIZER_CLASS_NAME: kafka.security.authorizer.AclAuthorizer
    KAFKA_ALLOW_EVERYONE_IF_NO_ACL_FOUND: "true"
  depends_on:
    - zoo1

```

images/kluster_producer.png

Figure 7: Cluster Kafka

Dopo di che si è andati a creare un topic in **Apache Kafka** chiamato **druid** e si è andati ad eseguire il producer appena creato.

```

sudo docker exec kafka1 kafka-topics --bootstrap-server kafka1:9092 --create
--topic druid

```

- Dopo di che siamo andati a creare un consumatore di eventi in python in grado di consumare gli eventi provenienti dal quel topic.

```
from json import loads
from kafka import KafkaConsumer
import json

consumer = KafkaConsumer(
    'druid',
    bootstrap_servers = ['kafka1:9092', 'kafka2:9093', 'kafka3:9094'],
    auto_offset_reset = 'earliest',
    enable_auto_commit = True,
    group_id = 'my-group',
    value_deserializer = lambda x : json.loads(x.decode('utf-8'))
)
for message in consumer:
    message = message.value
    print(message)
```

- Dopo di che per eseguire il producer e il consumatore si è andati ad utilizzare i seguenti comandi su due terminali distinti

```
python3 producer.py
python3 consumer.py
```

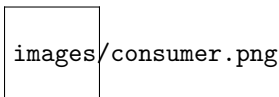


Figure 8: Iniezione dei dati

11.3 Creazione di una data pipeline con Apache Kafka, Apache Druid e Docker Compose

Le note riportate in questa sezione illustrano la costruzione e un test di prestazione di una data pipeline containerizzata ed eseguita con **Docker Compose**.

11.3.1 Svolgimento

- Come prima operazione si è andati a creare il seguente modello logico relazione, che riporta gli accessi effettuati da degli utenti

```
CREATE TABLE accessi(_time TIMESTAMP, nome TEXT, cognome TEXT, indirizzo
    TEXT, città TEXT, stato TEXT, cap INT,
    email TEXT, telefono TEXT, età INT, altezza DECIMAL(5,2), peso
    DECIMAL(5,2), reddito DECIMAL(6,2),
    datan DATE, professione TEXT, istruzione TEXT, hobby TEXT, nfigli INT,
    codice_cliente INT, datareg TIMESTAMP);
```

- Su tale modello dati si è andati a sviluppare il seguente generatore di eventi

```

from time import sleep
from kafka import KafkaProducer
import json
import random
import csv
from faker import Faker
import datetime
producer = KafkaProducer(
    bootstrap_servers = ["localhost:29092"],
    value_serializer = lambda x: json.dumps(x).encode("utf-8")
)
print("Connect succefully")
fake = Faker()
max=random.randint(4,20)
nomi= [fake.first_name() for _ in range(max)]
max=random.randint(4,20)
cognomi= [fake.last_name() for _ in range(max)]
max=random.randint(4,20)
indirizzi= [fake.address() for _ in range(max)]
#la libreria faker genera indirizzi con \n, che vanno sostituiti con uno
spazio
for i in range(len(indirizzi)):
    indirizzi[i]=indirizzi[i].replace('\n', ' ')
max=random.randint(4,20)
_citta=[fake.city() for _ in range(max)]
max=random.randint(4,20)
stati=[fake.country() for _ in range(max)]
max=random.randint(4,20)
_cap=[fake.zipcode() for _ in range(max)]
max=random.randint(4,20)
_email=[fake.email() for _ in range(max)]
max=random.randint(4,20)
telefoni=[fake.phone_number() for _ in range(max)]
max=random.randint(4,20)
_eta= [random.randint(18, 89) for _ in range(max)]
max=random.randint(4,20)
altezze= [round(random.uniform(120, 210), 2) for _ in range(max)]
max=random.randint(4,20)
pesi= [round(random.uniform(30, 180), 2) for _ in range(max)]
max=random.randint(4,20)
redditi= [round(random.uniform(1000, 10000), 2) for _ in range(max)]
max=random.randint(4,20)
daten= [fake.date_of_birth(minimum_age=18,
    maximum_age=89).strftime("%Y-%m-%d") for _ in range(max)]
max=random.randint(4,20)
professioni= [fake.job() for _ in range(max)]
max=random.randint(4,20)
_nfigli= [random.randint(0, 5) for _ in range(max)]

```

```

max=random.randint(4,20)
codici_cliente=[fake.random_number(digits=6) for _ in range(max)]
max=random.randint(4,20)
datereg= [fake.date_time_between(start_date="-1y",
    end_date="now").strftime("%Y-%m-%d %H:%M:%S") for _ in range(max)]
for n in range(500):
    for j in range(10000):
        nome= random.choice(nomi)
        cognome= random.choice(cognomi)
        indirizzo= random.choice(indirizzi)
        citta= random.choice(_citta)
        stato= random.choice(stati)
        cap= random.choice(_cap)
        email= random.choice(_email)
        telefono= random.choice(telefoni)
        eta= random.choice(_eta)
        altezza= random.choice(altezze)
        peso= random.choice(pesi)
        reddito= random.choice(redditi)
        datan= random.choice(datan)
        professione= random.choice(professioni)
        istruzione= fake.random_element(elements=("Scuola Secondaria",
            "Laurea triennale", "Laurea Magistrale", "Dottorato"))
        hobby= fake.random_element(elements=("Leggere", "Viaggiare", "Giocare
            a calcio", "Giocare ai videogiochi", "Fare sport"))
        nfigli= random.choice(_nfigli)
        codice_cliente= random.choice(codici_cliente)
        datareg= random.choice(datereg)
        accesso= datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
        my_data = {"Accesso": accesso, "nome": nome, "cognome": cognome,
            "indirizzo": indirizzo, "citta": citta, "stato": stato, "cap":
                cap,
            "email": email, "telefono": telefono, "eta": eta, "altezza":
                altezza, "peso": peso,
            "reddito": reddito, "datan": datan, "professione": professione,
            "istruzione": istruzione,
            "hobby": hobby,
            "nfigli": nfigli, "codice_cliente": codice_cliente,
            "datareg": datareg,
        }
        producer.send("accessi", value = my_data)
        sleep(1)
print("End")

```

È possibile notare che è stata applicata una randomici da 4 a 20 sui possibili valori che un attributo può assumere

- Successivamente è stati creati i servizi di **Apache Druid** andando a specificare una *docker network* che permetterà ai container di comunicare.

```

version: "2.2"

networks:
  kafka-druid:
    name: kafka-druid
    driver: bridge
    external: true

volumes:
  metadata_data: {}
  middle_var: {}
  historical_var: {}
  broker_var: {}
  coordinator_var: {}
  router_var: {}
  druid_shared: {}

services:
  postgres:
    container_name: postgres
    image: postgres:latest
    networks:
      - kafka-druid
    ports:
      - "5432:5432"
    volumes:
      - metadata_data:/var/lib/postgresql/data
    environment:
      - POSTGRES_PASSWORD=FoolishPassword
      - POSTGRES_USER=druid
      - POSTGRES_DB=druid

  zookeeper:
    container_name: zookeeper
    hostname: zookeeper
    image: confluentinc/cp-zookeeper:7.4.0
    networks:
      - kafka-druid
    ports:
      - "2181:2181"
    environment:
      - ZOOKEEPER_SERVER_ID=1
      - ZOOKEEPER_CLIENT_PORT=2181

  coordinator:
    image: apache/druid:26.0.0
    container_name: coordinator
    networks:

```

```

    - kafka-druid
volumes:
  - druid_shared:/opt/shared
  - coordinator_var:/opt/druid/var
depends_on:
  - zookeeper
  - postgres
ports:
  - "8081:8081"
command:
  - coordinator
env_file:
  - environment

broker:
  image: apache/druid:26.0.0
  container_name: broker
  networks:
    - kafka-druid
  volumes:
    - broker_var:/opt/druid/var
  depends_on:
    - zookeeper
    - postgres
    - coordinator
  ports:
    - "8082:8082"
  command:
    - broker
  env_file:
    - environment

historical:
  image: apache/druid:26.0.0
  container_name: historical
  networks:
    - kafka-druid
  volumes:
    - druid_shared:/opt/shared
    - historical_var:/opt/druid/var
  depends_on:
    - zookeeper
    - postgres
    - coordinator
  ports:
    - "8083:8083"
  command:
    - historical
  env_file:

```

```

- environment

middlemanager:
  image: apache/druid:26.0.0
  container_name: middlemanager
  networks:
    - kafka-druid
  volumes:
    - druid_shared:/opt/shared
    - middle_var:/opt/druid/var
  depends_on:
    - zookeeper
    - postgres
    - coordinator
  ports:
    - "8091:8091"
    - "8100-8105:8100-8105"
  command:
    - middleManager
  env_file:
    - environment

router:
  image: apache/druid:26.0.0
  container_name: router
  networks:
    - kafka-druid
  volumes:
    - router_var:/opt/druid/var
  depends_on:
    - zookeeper
    - postgres
    - coordinator
  ports:
    - "8888:8888"
  command:
    - router
  env_file:
    - environment

```

- In seguito si è andato a configurare il relativo file di configurazione di **Apache Druid**

```

#
# Licensed to the Apache Software Foundation (ASF) under one
# or more contributor license agreements. See the NOTICE file
# distributed with this work for additional information
# regarding copyright ownership. The ASF licenses this file
# to you under the Apache License, Version 2.0 (the
# "License"); you may not use this file except in compliance
# with the License. You may obtain a copy of the License at

```



```

#
# http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing,
# software distributed under the License is distributed on an
# "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
# KIND, either express or implied. See the License for the
# specific language governing permissions and limitations
# under the License.
#

# Java tuning
#DRUID_XMX=1g
#DRUID_XMS=1g
#DRUID_MAXNEWSIZE=250m
#DRUID_NEWSIZE=250m
DRUID_MAXDIRECTMEMORYSIZE=3072m
DRUID_SINGLE_NODE_CONF=nano-quickstart

druid_emitter_logging_logLevel=debug

druid_extensions_loadList=["druid-histogram", "druid-datasketches",
    "druid-lookups-cached-global", "postgresql-metadata-storage",
    "druid-multi-stage-query", "druid-kafka-indexing-service"]

druid_zk_service_host=zookeeper
druid_lookup_enableLookupSyncOnStartup=true
druid_lookup_lookupTierIsDatasource=false
druid_lookup_lookupTier=_default_tier

druid_broker_cache_useCache=true
druid_broker_cache_populateCache=true
druid_broker_cache_useResultLevelCache=true
druid_broker_cache_populateResultLevelCache=true
druid_cache_useCache=true
druid_cache_populateCache=true
druid_cache_useResultLevelCache=true
druid_cache_populateResultLevelCache=true
druid_metadata_storage_host=
druid_metadata_storage_type=postgresql
druid_metadata_storage_connector_connectURI=jdbc:postgresql://postgres:5432/druid
druid_metadata_storage_connector_user=druid
druid_metadata_storage_connector_password=FoolishPassword

druid_coordinator_balancer_strategy=cachingCost

druid_indexer_runner_javaOptsArray=["-server", "-Xmx1g", "-Xms1g",
    "-XX:MaxDirectMemorySize=3g", "-Duser.timezone=UTC",
    "-Dfile.encoding=UTF-8",

```

```

    "-Djava.util.logging.manager=org.apache.logging.log4j.jul.LogManager"]
druid_indexer_fork_property_druid_processing_buffer_sizeBytes=256MiB

druid_storage_type=local
druid_storage_storageDirectory=/opt/shared/segments
druid_indexer_logs_type=file
druid_indexer_logs_directory=/opt/shared/indexing-logs

druid_processing_numThreads=1
druid_processing_numMergeBuffers=1

DRUID_LOG4J=<?xml version="1.0" encoding="UTF-8" ?><Configuration
    status="WARN"><Appenders><Console name="Console"
    target="SYSTEM_OUT"><PatternLayout pattern="%d{ISO8601} %p [%t] %c -
    %m%n"/></Console></Appenders><Loggers><Root level="info"><AppenderRef
    ref="Console"/></Root><Logger name="org.apache.druid.jetty.RequestLog"
    additivity="false" level="DEBUG"><AppenderRef
    ref="Console"/></Logger></Loggers></Configuration>

```

- In seguito per mandare in esecuzione i container

```

|| docker compose -f nome_file up -d

```

Dopo tale operazione tutti i container relativi ad **Apache Druid** saranno in esecuzione

- Ora si configurano i container relativi ad **Apache Kafka**, per semplicità utilizzeremo un solo nodo come **Broker** e riutilizzeremo il nodo **Zookeeper** generato precedentemente come servizio di configurazione dei nodi

```

networks:
  kafka-druid:
    name: kafka-druid
    driver: bridge
    external: true

services:
  kafka:
    image: confluentinc/cp-kafka:7.4.0
    hostname: kafka
    container_name: kafka
    networks:
      - kafka-druid
    ports:
      - "29092:29092"
    environment:
      KAFKA_ADVERTISED_LISTENERS:
        INTERNAL://kafka:9092,EXTERNAL://localhost:29092
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP:
        INTERNAL:PLAINTEXT,EXTERNAL:PLAINTEXT
      KAFKA_INTER_BROKER_LISTENER_NAME: INTERNAL

```

```

KAFKA_ZOOKEEPER_CONNECT: "zookeeper:2181"
KAFKA_BROKER_ID: 1
KAFKA_LOG4J_LOGGERS:
    "kafka.controller=INFO,kafka.producer.async.DefaultEventHandler=INFO,
state.change.logger=INFO"
KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
KAFKA_TRANSACTION_STATE_LOG_REPLICATION_FACTOR: 1
KAFKA_TRANSACTION_STATE_LOG_MIN_ISR: 1
KAFKA_AUTHORIZER_CLASS_NAME: kafka.security.authorizer.AclAuthorizer
KAFKA_ALLOW_EVERYONE_IF_NO_ACL_FOUND: "true"

```

Dopo di che sia il nodo di **Apache Kafka**, sia quello di **Apache Druid** saranno in esecuzione.

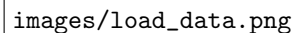
Per lanciare il generatore di eventi lanciare il comando

```

python3 nome_file.py

```

- in seguito per effettuare l'**injection** all'interno di **Apache Druid** usare il servizio gui **Load data - Streaming-Apache kafka** andando a specificare l'hostname e la porta esterna del **Broker** e il relativi **topic** di pubblicazione.



images/load_data.png

Figure 9: Load data - Streaming-Apache kafka

11.3.2 Confronto tra Apache Druid e Postgres

Per mettere in evidenza le capacità di questi strumenti andiamo ad utilizzare un datasource da 5000000 record aventi il seguente schema relazione.

Per eseguire i comandi direttamente all'interno del container di **PostgreSQL** eseguire il comando.

```
|| sudo docker exec postgres -ti bash

|| CREATE TABLE accessi( nome text, cognome text, indirizzo text, citta text, stato
||     text, cap int,
|| email text, telefono text, eta int, altezza decimal(5,2), peso decimal(5,2),
||     reddito decimal(6,2),
|| datan date, professione text, istruzione text, hobby text, nfigli int,
||     codice_cliente int, datareg timestamp, __time timestamp);
```

Utilizzando i seguenti parametri di configurazione per **Apache Druid** senza l'utilizzo della cache:

1. granularity segment per day ;
2. 800000 max rows in memory.

Inoltre per consentire l'utilizzo degli stessi dati sia in **Apache Druid** che in **PostgreSQL**, il generatore degli eventi viene modificato in modo tale da poter memorizzare gli eventi generati all'interno di un file *.csv.

```

from time import sleep
from kafka import KafkaProducer
import json
import random
import csv
from faker import Faker
import datetime
producer = KafkaProducer(
    bootstrap_servers = ["localhost:29092"],
    value_serializer = lambda x: json.dumps(x).encode("utf-8")
)
print("Connect succefully")
fake = Faker()
max=random.randint(4,20)
nomi= [fake.first_name() for _ in range(max)]
max=random.randint(4,20)
cognomi= [fake.last_name() for _ in range(max)]
max=random.randint(4,20)
indirizzi= [fake.address() for _ in range(max)]
#la libreria faker genera indirizzi con \n, che vanno sostituiti con uno spazio
for i in range(len(indirizzi)):
    indirizzi[i]=indirizzi[i].replace('\n', ' ')
max=random.randint(4,20)
_citta=[fake.city() for _ in range(max)]
max=random.randint(4,20)
stati=[fake.country() for _ in range(max)]
max=random.randint(4,20)
_cap=[fake.zipcode() for _ in range(max)]
max=random.randint(4,20)
_email=[fake.email() for _ in range(max)]
max=random.randint(4,20)
telefoni=[fake.phone_number() for _ in range(max)]
max=random.randint(4,20)
_eta= [random.randint(18, 89) for _ in range(max)]
max=random.randint(4,20)
altezze= [round(random.uniform(120, 210), 2) for _ in range(max)]
max=random.randint(4,20)
pesi= [round(random.uniform(30, 180), 2) for _ in range(max)]
max=random.randint(4,20)
redditi= [round(random.uniform(1000, 10000), 2) for _ in range(max)]
max=random.randint(4,20)
daten= [fake.date_of_birth(minimum_age=18, maximum_age=89).strftime("%Y-%m-%d")
    for _ in range(max)]
max=random.randint(4,20)
professioni= [fake.job() for _ in range(max)]
max=random.randint(4,20)
_nfigli= [random.randint(0, 5) for _ in range(max)]
max=random.randint(4,20)
codici_cliente=[fake.random_number(digits=6) for _ in range(max)]

```

```

max=random.randint(4,20)
datereg= [fake.date_time_between(start_date="-1y",
    end_date="now").strftime("%Y-%m-%d %H:%M:%S") for _ in range(max)]
volume=[]
for n in range(500):
    for j in range(10000):
        nome= random.choice(nomi)
        cognome= random.choice(cognomi)
        indirizzo= random.choice(indirizzi)
        citta= random.choice(_citta)
        stato= random.choice(stati)
        cap= random.choice(_cap)
        email= random.choice(_email)
        telefono= random.choice(telefoni)
        eta= random.choice(_eta)
        altezza= random.choice(altezze)
        peso= random.choice(pesi)
        reddito= random.choice(redditi)
        datan= random.choice(daten)
        professione= random.choice(professioni)
        istruzione= fake.random_element(elements=("Scuola Secondaria", "Laurea
            triennale", "Laurea Magistrale", "Dottorato"))
        hobby= fake.random_element(elements=("Leggere","Viaggiare","Giocare a
            calcio","Giocare ai videogiochi","Fare sport"))
        nfigli= random.choice(_nfigli)
        codice_cliente= random.choice(codici_cliente)
        datareg= random.choice(datereg)
        accesso= datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
        my_data = {"Accesso": accesso, "nome": nome, "Cognome": cognome,
            "indirizzo": indirizzo, "citta": citta, "stato": stato, "cap": cap,
            "email": email, "telefono": telefono, "eta": eta, "altezza": altezza,
            "peso": peso,
            "reddito":reddito, "datan": datan, "professione": professione,
            "istruzione": istruzione,
            "hobby": hobby,
            "nfigli": nfigli, "codice_cliente": codice_cliente,
            "datareg": datareg,
        }
        producer.send("accessi", value = my_data)
        element=[nome, cognome, indirizzo, citta, stato, cap, email, telefono,
            eta, altezza, peso, reddito, datan, professione, istruzione, hobby,
            nfigli, codice_cliente, datareg, accesso]
        volume.append(element)
        sleep(1)
with open('data.csv', 'w', newline='') as file:
    writer = csv.writer(file, quoting=csv.QUOTE_NONNUMERIC, delimiter=',')
    writer.writerows(volume)

print("End")

```

Per importare i dati contenuti nel file *.csv all'interno di **PostgreSQL** utilizzare il seguente comando.

```
COPY accessi(nome, cognome, indirizzo, citta, stato, cap,
email, telefono, eta, altezza, peso, reddito,
datan, professione, istruzione, hobby, nfigli, codice_cliente, datareg, __time)
From '/data.csv'
Delimiter ','
csv header;
```

Una volta effettuato l'**injection** di dati su entrambi i sistemi per misurare le performance si eseguiranno le seguenti query.

```
- query 1: SELECT DATE_TRUNC('day', __time), citta, COUNT(*)
           FROM accessi
           GROUP BY DATE_TRUNC('day', __time), citta

- query 2: SELECT stato, AVG(eta), AVG(reddito)
           FROM accessi
           GROUP BY stato

- query 3: SELECT DATE_TRUNC('year', __time), stato, professione, istruzione,
           nfigli, COUNT(*)
           FROM accessi
           WHERE nfigli > 0
           GROUP BY DATE_TRUNC('year', __time), stato, professione, istruzione,
           nfigli
           ORDER BY 5 DESC

- query 4: SELECT DATE_TRUNC('hour', __time), citta, AVG(eta)
           FROM accessi
           GROUP BY DATE_TRUNC('hour', __time), citta

- query 5: SELECT DATE_TRUNC('year', __time), DATE_TRUNC('month', __time),
           DATE_TRUNC('day', __time), stato, professione, istruzione, nfigli, COUNT(*)
           FROM accessi
           GROUP BY DATE_TRUNC('year', __time), DATE_TRUNC('month', __time),
           DATE_TRUNC('day', __time), stato, professione, istruzione, nfigli
           ORDER BY 5 DESC

- query 6: SELECT DATE_TRUNC('year', __time), DATE_TRUNC('month', __time),
           DATE_TRUNC('day', __time),
           DATE_TRUNC('hour', __time), stato, professione, istruzione, nfigli, COUNT(*)
           FROM accessi
           GROUP BY DATE_TRUNC('year', __time), DATE_TRUNC('month', __time),
           DATE_TRUNC('day', __time), DATE_TRUNC('hour', __time), stato,
           professione, istruzione, nfigli
           ORDER BY 5 DESC
```

	Apache Druid [s]	PostgreSQL [s]
query1	0.6	1.2
query2	0.2	1.9
query3	1.2	2
query4	0.6	1.1
query5	1.05	2.8
query6	1.30	3.3

Conclusioni Dalla seguente sperimentazione si evince una chiara superiorità di **Apache Druid** di un ordine di grandezza su un comune database relazione come **PostgreSQL**, in particolare nell'esecuzione di query che coinvolgono raggruppamenti, anche su intervalli temporali (ora, giorno, mese, anno).

11.3.3 Confronto tra Apache Druid con e senza rollup

Per mettere in evidenza la funzionalità di **rollup** si va a confrontare le prestazioni che **Apache Druid** sviluppa in un datasource da 5000000 di dati con e senza **rollup**. Per completezza verrà anche fatto un confronto con un classico database relazione come **PostgreSQL**.

Lo schema relazione è il seguente:

```
CREATE TABLE accessi(__time timestamp, nome text, cognome text, citta text, stato
text, datan date, istruzione text, hobby text);
```

Utilizzando i seguenti parametri per **Apache Druid** senza l'utilizzo della cache:

- granularity segment per day;
- 800000 max rows in memory;
- **roll up** on (per il datasource con **roll up**);
- query granularity **hour**.

Per consentire l'utilizzo degli stessi dati anche in **PostgreSQL** verrà generato un file *.csv contenente tutti i dati generati.

Il generatore dei dati è il seguente:

```
from kafka import KafkaProducer
import json
import random
import csv
from faker import Faker
import datetime
producer = KafkaProducer(
    bootstrap_servers = ["localhost:29092"],
    value_serializer = lambda x: json.dumps(x).encode("utf-8")
)
```



```

print("Connect succesfully")
fake = Faker()
locazione=[[fake.city(), fake.country()] for _ in range(200)]

utenti=list()
for i in range(5000):
    a=locazione[random.randint(0,199)]
    citta=a[0]
    stato=a[1]
    utenti.append([fake.first_name(),
                    fake.last_name(),fake.date_of_birth(minimum_age=18,
                    maximum_age=89).strftime("%Y-%m-%d"), citta, stato,
                    fake.random_element(elements=("Scuola Secondaria", "Laurea triennale",
                    "Laurea Magistrale", "Dottorato")),
                    fake.random_element(elements=("Leggere","Viaggiare","Giocare a
                    calcio","Giocare ai videogiochi","Fare sport")) ] )

volume=[]
for n in range(50):

    for j in range(100000):
        accesso=datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
        a=random.randint(0,4999)
        nome=utenti[a][0]
        cognome=utenti[a][1]
        datan=utenti[a][2]
        citta=utenti[a][3]
        stato=utenti[a][4]
        istruzione=utenti[a][5]
        hobby=utenti[a][6]
        my_data = {"accesso": accesso, "nome": nome, "cognome": cognome, "datan":
                    datan, "citta": citta, "stato": stato, "istruzione": istruzione,
                    "hobby": hobby
                    }
        producer.send("test_rollup11", value = my_data)
        element=[nome, cognome, citta, stato, datan,istruzione, hobby,accesso]
        volume.append(element)

with open('data.csv', 'w', newline='') as file:
    writer = csv.writer(file, quoting=csv.QUOTE_NONNUMERIC, delimiter=',')
    writer.writerows(volume)

print("End")

```

Una volta effettuata l'**injection** dei dati verranno effettuate le seguenti query per misurare le performance richieste.

```

- query 1: SELECT DATE_TRUNC('day', __time), citta, COUNT(*)
           FROM accessi
           GROUP BY DATE_TRUNC('day', __time), citta

```

```

- query 2: SELECT DATE_TRUNC('year', __time), DATE_TRUNC('month', __time), stato,
            COUNT(*)
            FROM accessi
            GROUP BY DATE_TRUNC('year', __time), DATE_TRUNC('month', __time), stato
            ORDER BY 4 DESC

- query 3: SELECT DATE_TRUNC('year', __time), DATE_TRUNC('month', __time),
            DATE_TRUNC('day', __time), stato, citta, COUNT(*)
            FROM accessi
            GROUP BY DATE_TRUNC('year', __time), DATE_TRUNC('month', __time),
                     DATE_TRUNC('day', __time), stato, citta
            ORDER BY 5 DESC

- query 4: SELECT DATE_TRUNC('year', __time), DATE_TRUNC('month', __time),
            DATE_TRUNC('day', __time), DATE_TRUNC('hour', __time), stato, citta, COUNT(*)
            FROM accessi
            GROUP BY DATE_TRUNC('year', __time), DATE_TRUNC('month', __time),
                     DATE_TRUNC('day', __time), DATE_TRUNC('hour', __time), stato, citta
            ORDER BY 5 DESC

```

Svolgimento

- Per attivare la funzionalità di **rollup** come prima operazione effettuare linjection dei in **Apache Druid**
- In *configureschema* andare a specificare il campo **rollup** su **true** e il campo **granularity** su **hour**



Figure 10: Attivazione della funzionalità di rollup

	Apache Druid con rollup[s]	Apache Druid senza rollup [s]	PostgreSQL [s]
query1	0.3	0.7	0.8
query2	0.4	0.75	1.1
query3	0.25	0.74	1.5
query4	0.16	0.8	1.6

Conclusioni Come riportato dalla tabella sovrastante, le misure effettuate sul datasource con **rollup** sono migliori rispetto a quelle effettuate senza **rollup**, questo perchè grazie a tale funzionalità si riduce il numero di record da processare. Seppur la cardinalità dei dati non sia così considerevole, le prestazioni mostrano già una diminuzione di un decimo dal punto di vista delle prestazioni di esecuzione (si passa da 5000000 a 150000 record).

11.3.4 Utilizzo delle tabelle di lookup in Apache Druid

Le note riportate in questa sezione si riferiscono alla creazione e utilizzo delle tabelle di **lookup** in un classico scenario di **data enrichment**. Il modello relazione dei dati generati sarà il seguente:

```
|| CREATE TABLE accessi(__time timestamp, citta text, stato text, codice_cliente);
```

La tabella di verrà utilizzata per andare a sostituire al *codice_cliente* il *nome* e il *cognome* del cliente.

Il generatore degli eventi è il seguente:

```
from kafka import KafkaProducer
import json
import random
import csv
from faker import Faker
import datetime

producer = KafkaProducer(
    bootstrap_servers = ["localhost:29092"],
    value_serializer = lambda x: json.dumps(x).encode("utf-8")
)

print("Connect successfully")
fake = Faker()
locazione=[[fake.city(), fake.country()] for _ in range(200)]

utenti=list()
for i in range(150):
    a=locazione[random.randint(0,99)]
    citta=a[0]
    stato=a[1]
    a=random.randint(0,999)
    utenti.append([a, fake.first_name(),
        fake.last_name(),fake.date_of_birth(minimum_age=18,
        maximum_age=89).strftime("%Y-%m-%d"), citta, stato] )

for i in range(150):
    print('\'+str(utenti[i][0])+\'\'+"':"+\'\''+utenti[i][1]+' '+utenti[i][2]+\'\'')
volume=[]
for n in range(500):
    accesso=datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    codice=utenti[random.randint(0,149)][0]
    nome=utenti[random.randint(0,149)][1]
    cognome=utenti[random.randint(0,149)][2]
```

```

datan=utenti[random.randint(0,149)][3]
citta=utenti[random.randint(0,149)][4]
stato=utenti[random.randint(0,149)][5]
my_data = {"accesso": accesso, "codice_cliente": str(codice), "datan":
    datan, "citta": citta, "stato": stato,
}
producer.send("lookup", value = my_data)
element=[codice, nome, cognome, citta, stato, datan, accesso]
volume.append(element)
with open('data.csv', 'w', newline='') as file:
    writer = csv.writer(file, quoting=csv.QUOTE_NONNUMERIC, delimiter=',')
    writer.writerows(volume)
print("End")

```

Una volta effettuata l'**injection** dei dati, si va a creare la tabella di **lookup** chiamata *tabella1*.

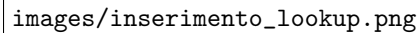


Figure 11: Creazione della tabella di lookup

In seguito si è andati ad eseguire la seguente query per verificare il corretto funzionamento della tabella di **lookup**.

```

SELECT LOOKUP(codice_cliente,'tabella1'), datan, citta, stato
FROM accessi

```



Figure 12: Risultato della query utilizzando la tabella di lookup

12 Test

12.1 High Availability

Per andare a testare l'alta affidabilità, si è andati a creare un cluster di nodi formato da: 1 nodo Zookeeper, 3 nodi broker (Kafka1, Kafka2, Kafka3) e si è andati ad istanziare un produttore e consumatore collegati al broker Kafka1. Ora per testare l'alta affidabilità basta simulare un malfunzionamento e verificare se i messaggi che il broker Kafka1 non è riuscito ad inviare al consumatore sono accessibili attraverso i server Kafka2 e Kafka3.

12.1.1 Svolgimento

- Come prima operazione si è andati a creare un cluster di 3 nodi Kafka e un nodo ZooKeeper utilizzando Docker Compose, utilizzando il comando

```
|| sudo docker compose -f nome_file_Kafka.yml up -d
```

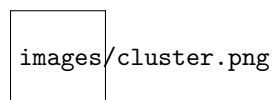
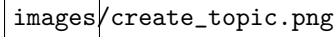


Figure 13: Creazione del cluster

- Creazione del topic "test".

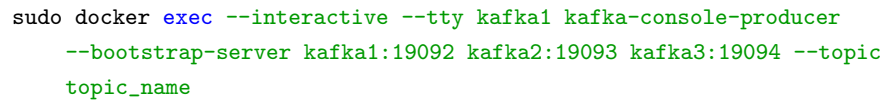
```
|| sudo docker exec kafka1 kafka-topics --bootstrap-server kafka1:19092 --create
--topic topic_name
```



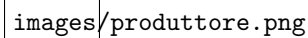
```
images/create_topic.png
```

Figure 14: Creazione del topic

- Creazione di produttore e consumatore.

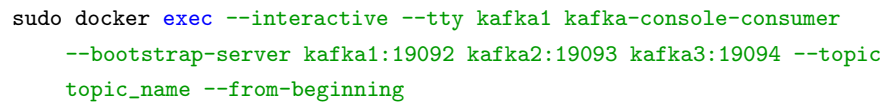


```
sudo docker exec --interactive --tty kafka1 kafka-console-producer
--bootstrap-server kafka1:19092 kafka2:19093 kafka3:19094 --topic
topic_name
```

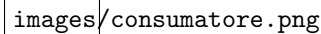


```
images/produttore.png
```

Figure 15: Creazione del produttore



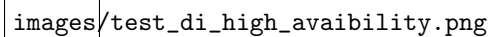
```
sudo docker exec --interactive --tty kafka1 kafka-console-consumer
--bootstrap-server kafka1:19092 kafka2:19093 kafka3:19094 --topic
topic_name --from-beginning
```



```
images/consumatore.png
```

Figure 16: Creazione del consumatore

- Se il nodo kafka1 viene spento, il produttore e consumatore possono continuare a comunicare collegandosi o al nodo kafka2 o al nodo kafka3.



```
images/test_di_high_avaibility.png
```

Figure 17: Test di alta affidabilità

12.2 Verifica di connessione al container da un altro host

Per andare a verificare la connessione al container a partire da un altro host non facciamo altro che connetterci al container attraverso la porta pubblica che espone, configurata nel parametro external di configurazione.

12.2.1 Svolgimento

- Come prima operazione si è andati a creare un cluster di 3 nodi Kafka e un nodo ZooKeeper utilizzando Docker Compose.

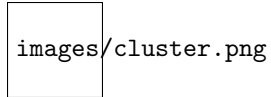


Figure 18: Creazione del cluster

- Dopo di che abbiamo creato dei topic da esempio.

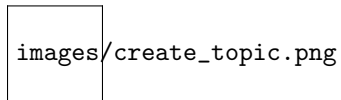


Figure 19: Creazione dei topic

- Ed infine ci siamo collegati al cluster a partire da un altro host (localhost) per visualizzare la lista dei topic creati.

```
|| bin/kafka-topics.sh --list --bootstrap-server localhost:9093
```

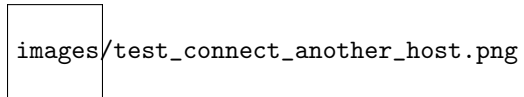


Figure 20: Connessione da altro host